



TECHNISCHE
UNIVERSITÄT
DARMSTADT

ADVANCES IN ILP-BASED MODULO SCHEDULING FOR HIGH-LEVEL SYNTHESIS

Vom Fachbereich Informatik der
Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)
von

Dipl.-Inform. Julian Oppermann
aus Göttingen

Referenten: Prof. Dr.-Ing. Andreas Koch
Assoc. Prof. Oliver Sinnen

Tag der Prüfung: 30.10.2019

D 17
Darmstadt, 2019

Julian Oppermann: *Advances in ILP-based Modulo Scheduling for High-Level Synthesis*.
Dissertation. Technische Universität Darmstadt. 2019.

Bitte zitieren Sie dieses Dokument als:

URN: urn:nbn:de:tuda-tuprints-92720

URL: <https://tuprints.ulb.tu-darmstadt.de/id/eprint/9272>

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt.

<https://tuprints.ulb.tu-darmstadt.de>

tuprints@ulb.tu-darmstadt.de

Die Veröffentlichung steht unter folgender Creative Commons-Lizenz:

Attribution – NonCommercial – NoDerivatives 4.0 International (CC BY-NC-ND 4.0)

<https://creativecommons.org/licenses/by-nc-nd/4.0/>



ERKLÄRUNGEN LAUT PROMOTIONSORDNUNG

§8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 2019

Julian Oppermann

ABSTRACT

In today’s heterogenous computing world, field-programmable gate arrays (FPGA) represent the energy-efficient alternative to generic processor cores and graphics accelerators. However, due to their radically different computing model, automatic design methods, such as high-level synthesis (HLS), are needed to harness their full power. HLS raises the abstraction level to behavioural descriptions of algorithms, thus freeing designers from dealing with tedious low-level concerns, and enabling a rapid exploration of different microarchitectures for the same input specification. In an HLS tool, scheduling is the most influential step for the performance of the generated accelerator. Specifically, modulo schedulers enable a pipelined execution, which is a key technique to speed up the computation by extracting more parallelism from the input description.

In this thesis, we make a case for the use of integer linear programming (ILP) as a framework for modulo scheduling approaches. First, we argue that ILP-based modulo schedulers are practically usable in the HLS context. Secondly, we show that the ILP framework enables a novel approach for the automatic design of FPGA accelerators.

We substantiate the first claim by proposing a new, flexible ILP formulation for the modulo scheduling problem, and evaluate it experimentally with a diverse set of realistic test instances. While solving an ILP may incur an exponential runtime in the worst case, we observe that simple countermeasures, such as setting a time limit, help to contain the practical impact of outlier instances. Furthermore, we present an algorithm to compress problems before the actual scheduling.

An HLS-generated microarchitecture is comprised of operators, i.e. single-purpose functional units such as a floating-point multiplier. Usually, the allocation of operators is determined before scheduling, even though both problems are interdependent. To that end, we investigate an extension of the modulo scheduling problem that combines both concerns in a single model. Based on the extension, we present a novel multi-loop scheduling approach capable of finding the fastest microarchitecture that still fits on a given FPGA device – an optimisation problem that current commercial HLS tools cannot solve. This proves our second claim.

ZUSAMMENFASSUNG

Heutzutage werden komplexe Probleme in Wissenschaft und Technik von heterogenen Rechnersystemen gelöst. In diesem Umfeld haben sich sogenannte *field-programmable gate arrays* (FPGA) als energieeffizientere Alternative zum Rechnen auf normalen Prozessorkernen oder Grafikkarten etabliert. Bei FPGAs handelt es sich um Halbleiterbauteile, die beim Einschalten einen beliebigen Schaltkreis abbilden können, welcher dann die gewünschten Berechnungen durchführt. Da sich dieses "Programmiermodell" grundlegend von den Gewohnheiten der Software-Welt unterscheidet, sind die Anforderungen an die verwendeten Entwurfswerkzeuge hoch.

Die Synthese eines Schaltkreises aus einer Problembeschreibung in einer Hochsprache wie C wird *high-level synthesis* (HLS) genannt. HLS kann Hardware-Ingenieure merklich entlasten, weil viele konkrete Aspekte des Schaltkreises automatisch erzeugt werden. Dies ermöglicht auch eine schnelle Untersuchung verschiedener Entwurfsalternativen. In einem HLS-Werkzeug hat die Ablaufplanung (engl. *scheduling*) den größten Einfluss auf die Leistung des generierten Schaltkreises. Eine wichtige Technik, um die Ausführungsgeschwindigkeit weiter zu verbessern, ist die Fließbandverarbeitung (engl. *pipelining*) von Berechnungen. Diese wird durch das Lösen eines zyklischen Ablaufplanungsproblems (engl. *modulo scheduling*) ermöglicht.

Diese Dissertation untersucht den Einsatz von mathematischen Optimierungsmethoden, genauer von ganzzahliger linearer Optimierung (engl. *integer linear programming*, ILP), zur Lösung von Modulo Scheduling-Problemen. Es werden zwei Hauptthesen aufgestellt. Erstens, ILP-basierte Verfahren sind praktisch nutzbar im Kontext von HLS-Werkzeugen. Zweitens, ILP-basierte Verfahren eröffnen neue Möglichkeiten in der automatischen Synthese von Schaltkreisen.

Als Beleg für die erste These wird eine neue, flexible ILP-Formulierung des Scheduling-Problems vorgestellt und anhand einer Vielzahl von Problem-Instanzen aus der Praxis evaluiert. Einfache Maßnahmen wie das Setzen eines Zeitlimits helfen, die theoretisch exponentiellen Laufzeiten beim ILP-Lösen abzufangen. Außerdem wird ein Algorithmus zur Komprimierung des Problems vor der eigentlichen Ablaufplanung beschrieben.

Bisher ist es üblich, vor der Ablaufplanung zu bestimmen, welche und insbesondere wie viele sogenannte Operatoren im späteren Schaltkreis zur Verfügung stehen, um Teilberechnungen (z. B. eine Multiplikation von zwei Werten) durchzuführen. Tatsächlich sind aber die Ablaufplanung und Operatorallokation voneinander abhängige Probleme. Es wird daher eine Erweiterung des Modulo Scheduling-Problems vorgeschlagen, die beide Problemaspekte gemeinsam mo-

delliert. Darauf basierend wird ein Scheduling-Verfahren vorgestellt, welches die schnellste Schaltung findet, die gerade noch auf einen vorgegebenen FPGA passt – ein Optimierungsproblem, welches in der Form nicht direkt von kommerziellen HLS-Werkzeugen gelöst werden kann. Dies belegt die zweite These.

ACKNOWLEDGMENTS

Finishing a project as daunting as writing a doctoral thesis is not feasible without the help of others.

First of all, I am sincerely grateful to Andreas Koch for the supervision of my work, providing guidance in the world of academia, giving me the freedom to explore the ideas that eventually led to this thesis, and answering all the last-minute paper feedback requests.

I have been exceptionally lucky to have had a second mentor in Oliver Sinnen, who inspired me to be stubborn and curious, spent his free evenings in countless Skype meetings to discuss our research, and provided a desk in Auckland from time to time.

I would like to thank all my past and present colleagues at the Embedded Systems and Applications group in Darmstadt, for making a couple of offices filled with FPGA boards such a special place. I am especially thankful to have worked with Lukas Sommer, who was always available for discussions and a teammate on most of my projects, including being instrumental in getting the infamous eternal paper accepted.

It has been a great pleasure to collaborate with Patrick Sittel and Martin Kumm. Together, we brought back modulo scheduling to the agendas of our conferences!

I am very grateful to my parents, who made it possible for me to choose my way and follow my interests up to this point. Finally, I cannot imagine achieving any of this without my wife Melanie, who has been my interface to the Operations Research community, but more importantly, always supported me in this endeavour and kept pushing me forward as my “personal PostDoc”.

The experiments for this research were conducted on the Lichtenberg high-performance computing cluster at TU Darmstadt. I would like to thank Xilinx, Inc. for supporting my work by hardware and software donations.

CONTENTS

1	INTRODUCTION	1
1.1	Thesis Contributions	4
1.2	Thesis Outline	6
2	FOUNDATIONS	7
2.1	Common Components of High-Level Synthesis Flows .	7
2.1.1	Frontend	7
2.1.2	Operator Library	9
2.1.3	Algorithmic Steps	10
2.1.4	Controller Generation	11
2.1.5	Backend	11
2.2	Key High-Level Synthesis Techniques	11
2.2.1	Operator Chaining	11
2.2.2	Operator Sharing	12
2.2.3	Loop Pipelining	13
2.3	Modulo Scheduling	14
2.4	Formal Definition of the Modulo Scheduling Problem .	15
2.5	Scheduler Support for Operator Chaining	19
2.6	Bounds for the II Search Space	22
2.6.1	Lower Bound	22
2.6.2	Upper Bound	23
2.7	Survey of Modulo Scheduling Approaches	24
2.7.1	Target Architecture	24
2.7.2	Search Strategies for the II	25
2.7.3	Heuristic vs. Exact Approaches	26
2.8	Prior ILP Formulations	29
2.8.1	Formulation by Eichenberger and Davidson . .	29
2.8.2	Formulation by Šůcha and Hanzálek	31
2.9	Research Gap	32
3	EXACT AND PRACTICAL MODULO SCHEDULING	35
3.1	The MOOVAC Formulation	35
3.1.1	MOOVAC-S	36
3.1.2	MOOVAC-I	40
3.2	Strategies for Modulo Scheduling in Practice	42
3.2.1	Time-limited Scheduling	42
3.2.2	Bounded Schedule Length	44
3.3	Experimental Evaluation	47
3.3.1	Compiler Context	47
3.3.2	Reference Schedulers	47
3.3.3	Test Setup	48
3.3.4	Test Instances	48
3.3.5	Comparison of Approaches, Time Limits and Bounds	49

3.3.6	FPGA Implementation	60
3.4	Chapter Summary	60
4	DEPENDENCE GRAPH PREPROCESSING	61
4.1	Analysis	61
4.1.1	Instances	61
4.1.2	Exact Schedulers	62
4.1.3	Critical Operations	63
4.2	Modulo Scheduling with Compressed Problems	63
4.2.1	Construction of a Compressed Problem Instance	67
4.2.2	Modulo Scheduling	70
4.2.3	Schedule Completion	70
4.3	Experimental Evaluation	71
4.4	Chapter Summary	77
5	RESOURCE-AWARE MODULO SCHEDULING	79
5.1	Scheduling Framework	79
5.1.1	The Resource-Aware Modulo Scheduling Problem	80
5.1.2	The Multi-Objective RAMS Problem	82
5.1.3	Bounds	83
5.1.4	Trivial Allocation	84
5.2	Extension of Existing ILP Formulations	84
5.2.1	Formulation by Eichenberger and Davidson	85
5.2.2	Formulation by Šůcha and Hanzálek	85
5.2.3	Moovac formulation	85
5.3	Approaches for the MORAMS Problem	86
5.3.1	ε -Approach	86
5.3.2	Iterative Approach	88
5.3.3	Dynamic Lower Bound for the Allocation	90
5.4	Evaluation	90
5.5	Chapter Summary	94
6	SKYCASTLE: A RESOURCE-AWARE MULTI-LOOP SCHEDULER	95
6.1	Background	95
6.1.1	Motivational Example	96
6.1.2	Approach and Contributions	96
6.1.3	Related Work	97
6.1.4	As Part of the HLS Scheduler	97
6.1.5	Pipelining-focussed Exploration	98
6.1.6	General Design-Space Exploration	98
6.2	The Multi-Loop Scheduling Problem	99
6.2.1	Informal Problem Description	99
6.2.2	Extended Notation	100
6.2.3	Formal Problem Definition	101
6.2.4	Bounds	106
6.2.5	Compatibility with Vivado HLS	108
6.3	Extensions for MOOVAC	109
6.3.1	Alternative Modulo Decomposition	109
6.3.2	Alternative Operator Constraints	109

6.4	Resource-Aware Multi-Loop Scheduler	112
6.4.1	Precomputing Solutions	113
6.4.2	SKYCASTLE ILP Formulation	114
6.5	Case Studies	120
6.5.1	Kernels	120
6.5.2	Experimental Setup	123
6.5.3	Results	124
6.6	Chapter Summary	125
7	CONCLUSION	127
7.1	On the Practicality of ILP-based Modulo Scheduling	127
7.2	On the Flexibility of the MoovAC formulation	128
7.3	On New Design Methods for Hardware Accelerators	128
7.4	Outlook	129
	BIBLIOGRAPHY	131

LIST OF FIGURES

Figure 1.1	Extremely simplified on-chip structure of an FPGA	2
Figure 1.2	A 2-input LUT, implementing the XOR function. The bits in the column “out” are configurable.	2
Figure 1.3	Spatial computing using operators instantiated on the FPGA’s low-level resources	2
Figure 2.1	CDFG for code snippet in Listing 2.1. Let $c_1 = 0x08040201$, and $c_2 = 0x11111111$. Red edges denote inter-iteration dependences. Dotted edges represent memory dependences. . .	8
Figure 2.2	An operator allocation for the CDFG in Figure 2.1. The rectangle sizes hint at different resource demands for the operator types. . . .	10
Figure 2.3	A schedule for the CDFG in Figure 2.1	10
Figure 2.4	A binding, shown for a part of the CDFG in Figure 2.1	11
Figure 2.5	Application of operator chaining to the CDFG in Figure 2.1	12
Figure 2.6	A multiplication operator, shared between two operations	12
Figure 2.7	Loop pipelining with $II=5$, for the CDFG in Figure 2.1, with operator chaining. The CDFG constants have been omitted for clarity.	13
Figure 2.8	Additional chain-breaking edges (green, edge latency = 1) for the extended MSP defined by Table 2.3 and Table 2.5	21
Figure 2.9	Application of the pigeonhole principle to determine a lower bound for the II with regards to a shared operator type q with $b_q = 1$. Each q -instance provides II -many slots to accept inputs from different operations. If only one q instance is available (as shown), the II must therefore be at least 4 in order to accommodate operations A–D.	22
Figure 2.10	Now considering a shared operator type q with $b_q = 2$. Two q -instances together provide six slots, which is just enough to accommodate three operations blocking two slots each. A valid binding does exist, but will require unrolling the finished schedule [24].	23

Figure 3.1	MRT as induced by the operations' w - and m -decision variables. Each operator type instance can only be used by one operation per congruence class, thus each cell can be occupied by at most one operation.	38
Figure 3.2	Consider two operations i and j that compete for the same MRT cell as indicated in Figure 3.1. This sketch shows the values of the overlap variables (white = '0', grey = '1') for different assignments of w_j and m_j in relation to w_i and m_i . For example, in the top right corner, we assume $w_j = w_i - 1$ and $m_j = m_i + 1$, which is an assignment that does not result in a conflict.	38
Figure 3.3	Distribution of modulo schedule lengths, and their bounds. Note the logarithmic scale on the Y-axis.	45
Figure 3.4	Histogram of the number of loops according to the relative position of their Π^\bullet within the Π search space, i.e. $0 = \Pi^{\min}$ and $1 = \Pi^{\max}$	49
Figure 3.5	Performance profile of the scheduling times (5 minute time limit), showing the number of loops for which the scheduling time with a particular configuration is at most X times slower than the fastest scheduling time for each individual loop. The table shows the values for the special case $X = 1$, i.e. the number of loops for which a configuration defined the fastest scheduler runtime.	58
Figure 3.6	Maximum clock frequencies on Virtex-7 after HLS and place & route for the different schedulers (T^{IM} , 5 min configurations). Note that the Y-axis starts at 100 MHz.	59
Figure 4.1	Example instance	64
Figure 4.2	Compressed instance: critical operations	64
Figure 4.3	Compressed instance: constructed edges	65
Figure 4.4	Compressed instance: edges filtered, backedges added	65
Figure 4.5	Longest path length analysis results for all $j \in O$: showing the values stored in $\text{LPL}_j^{\text{IN}}[k]$ and $\text{LPL}_j^{\text{OUT}}[k]$, for reachable preceding critical operations $k \in O^{\text{Cr}}$. Cells marked with "-" contain the value $-\infty$	68
Figure 5.1	Different trade-offs regarding the throughput (smaller Π is better) and resource demand (fewer allocated operators is better)	80

Figure 5.2	Template model for resource-aware modulo scheduling	85
Figure 5.3	Trade-off points for instance <code>splin_pf</code> , computed with the iterative approach	92
Figure 6.1	An example instance of the multi-loop scheduling problem	99
Figure 6.2	Overview of MLS problem model	104
Figure 6.3	Interaction of the m , μ and χ variables in the alternative modelling of the operator constraints	110
Figure 6.4	Interaction of the m , μ and χ variables in the alternative modelling of the operator constraints, simplified for fully-pipelined operator types with a blocking time of 1	110
Figure 6.5	Flow of the SKYCASTLE scheduler	113
Figure 6.6	Outline of the SKYCASTLE ILP for the example instance from Figure 6.1. Not all decision variables are shown.	114
Figure 6.7	MLS problem structure for SPN kernel, and legend	121
Figure 6.8	MLS problem structure for FFT kernel	121
Figure 6.9	MLS problem structure for LULESH kernel	122

LIST OF TABLES

Table 2.1	Problem signature of the Modulo Scheduling Problem	15
Table 2.2	Supplementary notations	16
Table 2.3	MSP instance corresponding to the CDFG in Figure 2.1. Constants were omitted for brevity. A valid schedule is shown in Figure 2.7.	18
Table 2.4	Chaining extension for the Modulo Scheduling Problem	19
Table 2.5	Chaining subproblem example, for the MSP in Table 2.3	21
Table 3.1	MOOVAC-S: Decision variables	37
Table 3.2	Allocation for shared operator types	48
Table 3.3	Problem sizes	48
Table 3.4	Scheduling times for combinations of approaches, time limits and bounds	53
Table 3.5	Scheduling times for combinations of approaches, time limits and bounds (continued)	54
Table 3.6	Schedule quality for combinations of approaches, time limits and bounds	55

Table 3.7	Schedule quality for combinations of approaches, time limits and bounds (continued)	56
Table 3.8	Quality metrics for the computed solution S, as used in Tables 3.6 to 3.7	57
Table 4.1	Additional notation used in the problem-compressing approach	66
Table 4.2	Compression results: Problem size	73
Table 4.3	Compression results: ILP size	74
Table 4.4	Scheduling results: EDFORM	75
Table 4.5	Scheduling results: MOOVAC-S	76
Table 5.1	Problem signature modification for the Resource-Aware Modulo Scheduling problem	81
Table 5.2	Supplementary notations	81
Table 5.3	Problem sizes	90
Table 5.4	Design-space exploration results for 204 instances	93
Table 6.1	Notations to describe the MLS problem: Sets	102
Table 6.2	Notations to describe the MLS problem: Attributes	103
Table 6.3	Notations to describe the MLS problem: Mappings	104
Table 6.4	Bounds for the MLS problem	107
Table 6.5	Scheduling and system composition results	126
Table 7.1	Formulation capabilities	129

LISTINGS

Listing 2.1	An example loop	8
Listing 6.1	Sum-Product Network example	96

ACRONYMS

FPGA	Field-Programmable Gate Array
LUT	Look-Up Table
DSP	Digital Signal Processing Block
FF	Flip-Flop
BRAM	Block RAM

HLS	High-Level Synthesis
VLIW	Very-Long-Instruction-Word
LP	Linear Program
SDC	System of Difference Constraints
ILP	Integer Linear Program
CP	Constraint Programming
SAT	Boolean Satisfiability Problem
II	Initiation Interval
MRT	Modulo Reservation Table
MSP	Modulo Scheduling Problem
RAMS	Resource-Aware Modulo Scheduling
MORAMS	Multi-Objective Resource-Aware Modulo Scheduling
MLS	Multi-Loop Scheduling
IR	Intermediate Representation
SSA	Static Single Assignment
CDFG	Control-Data-Flow Graph
HDL	Hardware Design Language
CPU	Central Processing Unit
GPU	Graphics Processing Unit
SPN	Sum-Product Network
PGM	Probabilistic Graphical Model

INTRODUCTION

Field-Programmable Gate Arrays (FPGAs) offer flexible computing capabilities, which are used to solve challenging real-world problems in domains such as radio astronomy [89], computer vision [39], molecular simulations [82], model checking [10] and machine learning [58]. In today's heterogeneous and massively parallel computing world, they are often touted as the energy-efficient alternative to software-programmable processors and general-purpose computing on graphics processors. The interest is also not purely academic, as Amazon [3] and Microsoft [9] have started to use FPGAs in their data centres, and Apple announced to offer an FPGA-based video accelerator card for their upcoming workstation computer.

However, in contrast to CPUs and GPUs, which can be programmed with convenient frameworks and following common software development practices, FPGAs require a radically different development process to accommodate their internal structure.

FPGAs are semiconductor chips that provide programmable arrays of logic and memory elements, which are surrounded by a programmable interconnect. Figure 1.1 gives a simplified intuition. Typical logic elements are Look-Up Tables (LUTs), which are used to implement arbitrary Boolean functions (see Figure 1.2 for an example) and Digital Signal Processing Blocks (DSPs), which accelerate arithmetic functions. The simplest memory elements are Flip-Flops (FFs), which are often paired with LUTs. Kastner, Matai and Neuendorffer [46] give an overview on the basic FPGA technology.

At this lowest level, programming an FPGA means loading a stream of configuration bits onto the device, which sets up the low-level *resources* to perform their intended function, and establishes the connections between them. This process happens not only once in the factory, but every time the device is powered on – hence the name, “field-programmable”.

However, programming a useful accelerator of any kind at this level is intractable. Instead, a common abstraction is *spatial* computing, as illustrated in Figure 1.3. Several *operators* are instantiated on the device, using a certain amount of the low-level resources, and are connected to form a *datapath*. The computation progresses as the input and intermediate values flow through the datapath, and yield the result after the last operator finishes. Note that there is no (software) program of any kind being executed. Rather, the engineer designs a *microarchitecture*, specifically for the computation.

The largest FPGA devices available today exceed one billion configuration bits.

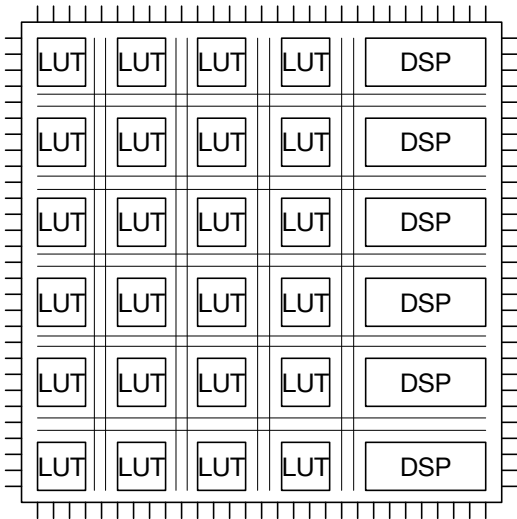


Figure 1.1: Extremely simplified on-chip structure of an FPGA

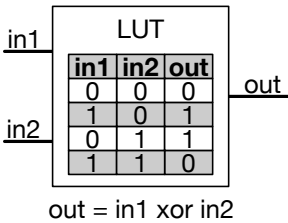


Figure 1.2: A 2-input LUT, implementing the XOR function. The bits in the column “out” are configurable.

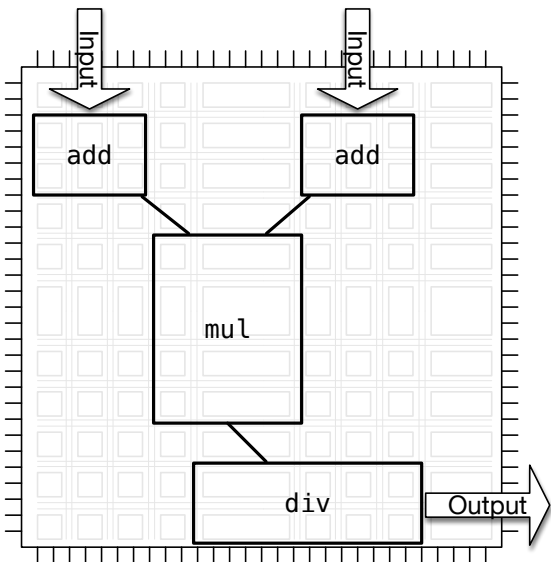


Figure 1.3: Spatial computing using operators instantiated on the FPGA's low-level resources

Coming up with such a design manually, and writing it down in terms of Hardware Design Languages (HDLs) such as Verilog and VHDL, is a tedious and error-prone process, and requires a lot of experience. It is a common understanding in the community that productive FPGA design requires better tools. Incidentally, none of the recently proposed accelerators mentioned above were developed in an HDL.

High-Level Synthesis (HLS) [14] raises the abstraction level for FPGA design. In the broadest sense, an HLS tool accepts a *behavioural*, un-timed description of the problem, and automatically generates a microarchitecture comprised of a synchronous datapath and a controller circuit to control the individual operators in it.

Many HLS tools accept some form of C code as input [61], arguably because of the language’s ubiquity in computer science. However, this choice fuelled expectations that HLS might be the holy grail to make FPGA accelerators accessible not only to hardware engineers, but to the much larger population of software programmers. Despite intensive research efforts by both the FPGA vendors and the academic community, this aim has not become reality yet.

From a compiler engineer’s viewpoint, it is relatively easy to construct a microarchitecture that performs a certain computation. It is, however, very hard to actually accelerate the computation compared to a modern software-programmable processor. FPGAs gain their energy-efficiency by operating at low frequencies (usually between 100–300 MHz), and therefore need to exploit as much parallelism as possible to bridge the “speed gap” to a single processor core, as even embedded CPUs operate at 1 GHz and more. The use of the C language is one part of the problem here, due to its sequential nature, which hinders the automatic discovery of parallel computations. Often, good HLS designs also require algorithmic changes, or rewrites of the C code to match certain idioms understood by the HLS tool.

Still, HLS is a valuable tool to make hardware engineers more productive, as it gives them the ability to experiment with different microarchitectures for the same input specification, in order to explore different trade-offs regarding the performance and resource demand, without the need to deal with low-level implementation details.

Furthermore, the FPGA community has successfully adopted parallel programming environments such as OpenCL [47] to accelerate complex, real-world scientific applications, e.g. [82, 89, 92]. These applications are typically comprised of multiple kernels in some form of processing pipeline. The OpenCL environment handles the data transfers from and to the accelerator and between the kernels. Crucially, the actual hardware implementing the kernels is constructed using standard HLS technology. Note that a single kernel implementation might be replicated tens or hundreds of times to leverage data paral-

lelism. Then, it is desirable to get the best solution possible – ideally, a provably optimal one. Investigating improvements to HLS tools is therefore as current and important as ever.

*A brief overview can
be found in
Section 2.1.*

While the development and maintenance of an HLS tool is a huge engineering effort at the intersection of compiler construction and hardware design, this thesis specifically addresses the core algorithmic steps that apply to any HLS flow, and in particular, the *scheduling* step: for the transformation from the untimed input description to the synchronous datapath, the operations that make up the computation need to be scheduled, i.e. a discrete start time is assigned to each of them. The scheduler is the main source of automatically discoverable parallelism during HLS, and therefore decisive for the performance of the generated microarchitecture. In this context, *loop pipelining* is a key optimisation technique for HLS from sequential input specifications: the partially overlapping execution of subsequent loop iterations intended to increase the accelerator’s throughput and the utilisation of the functional units within the datapath corresponding to the loop’s computation. To this end, new loop iterations are started after a fixed number of time steps, called the *Initiation Interval* (II).

Loop pipelining is usually enabled by modulo schedulers [74], which were first proposed and actively investigated in the 1990s, when Very-Long-Instruction-Word (VLIW) processor architectures were popular. More recently, due to their relevance to HLS, there has been a spike of interest in the academic community regarding HLS-centric modulo schedulers [6, 7, 17, 79, 84, 95], coinciding with the previous publication of parts of this thesis.

1.1 THESIS CONTRIBUTIONS

Modulo scheduling is a hard combinatorial problem. Especially during the VLIW period, modulo schedulers considered suitable for integration into a production compiler were *heuristic* algorithms. *Exact* approaches, capable of computing provably optimal schedules, were also developed, but usually deemed impractical because of their potentially longer runtimes.

We believe that the computing power available today, coupled with the advances in solver technology for Integer Linear Programs (ILPs), make it worthwhile to *be stubborn*, and challenge this old preconception. To this end, we advocate for ILP-based modulo scheduling because it ...

- makes the minimisation of the II as part of one linear program possible, which enables a novel strategy to tackle the Modulo Scheduling Problem (MSP),

- delivers optimal solutions for the majority of instances we encountered in our extensive experimental evaluation,
- still delivers good solutions with a known quality gap, when subjected to a limited time budget for particularly large instances, and
- enables resource-aware modulo scheduling, which combines the HLS scheduling and allocation steps, and thus allows the exploration of different microarchitectures during the mathematical optimisation process.

Our novel multi-loop scheduler, SKYCASTLE, leverages all these properties to solve the common and practically relevant design problem of finding the fastest microarchitecture that still fits within given resource constraints.

This thesis is based on the following peer-reviewed publications:

- [63] Julian Oppermann, Andreas Koch, Melanie Reuter-Oppermann and Oliver Sinnen. ‘ILP-based modulo scheduling for high-level synthesis’. In: *International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES, Pittsburgh, Pennsylvania, USA*. 2016
- [65] Julian Oppermann, Melanie Reuter-Oppermann, Lukas Sommer, Andreas Koch and Oliver Sinnen. ‘Exact and Practical Modulo Scheduling for High-Level Synthesis’. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETs)* 12.2 (2019)
- [66] Julian Oppermann, Melanie Reuter-Oppermann, Lukas Sommer, Oliver Sinnen and Andreas Koch. ‘Dependence Graph Preprocessing for Faster Exact Modulo Scheduling in High-Level Synthesis’. In: *International Conference on Field Programmable Logic and Applications, FPL Dublin, Ireland*. 2018
- [67] Julian Oppermann, Patrick Sittel, Martin Kumm, Melanie Reuter-Oppermann, Andreas Koch and Oliver Sinnen. ‘Design-Space Exploration with Multi-Objective Resource-Aware Modulo Scheduling’. In: *International Conference on Parallel and Distributed Computing, Euro-Par, Göttingen, Germany*. 2019
- [68] Julian Oppermann, Lukas Sommer, Lukas Weber, Melanie Reuter-Oppermann, Andreas Koch and Oliver Sinnen. ‘SkyCastle: A Resource-Aware Multi-Loop Scheduler for High-Level Synthesis’. In: *International Conference on Field-Programmable Technology, FPT*. 2019

The author was part of research efforts adjacent to the work presented in this thesis, which resulted in the following peer-reviewed publications:

- [69] Julian Oppermann, Sebastian Vollbrecht, Melanie Reuter-Oppermann, Oliver Sinnen and Andreas Koch. ‘GeMS: a generator for modulo scheduling problems: work in progress’. In: *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES, Torino, Italy*. 2018

- [79] Patrick Sittel, Martin Kumm, Julian Oppermann, Konrad Möller, Peter Zipf and Andreas Koch. ‘ILP-Based Modulo Scheduling and Binding for Register Minimization’. In: *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*. 2018
- [80] Patrick Sittel, Julian Oppermann, Martin Kumm, Andreas Koch and Peter Zipf. ‘HatScheT: A Contribution to Agile HLS’. In: *International Workshop on FPGAs for Software Programmers, FSP*. 2018

1.2 THESIS OUTLINE

CHAPTER 1 (this chapter) shows the importance of FPGA accelerators for modern heterogenous computing, the ongoing need for better HLS to help designers create such FPGA-based accelerators, and identifies modulo scheduling as a highly influential core algorithm in any HLS tool.

CHAPTER 2 introduces the fundamental HLS terminology, formally defines the Modulo Scheduling Problem (MSP), and surveys existing scheduling approaches.

CHAPTER 3 presents the Moovac formulation, which is the first ILP-based modulo scheduling formulation that integrates the search for the optimal ILP. We discuss and evaluate strategies to use an exact modulo scheduler in practice.

CHAPTER 4 describes and evaluates a compression algorithm for instances of the MSP.

CHAPTER 5 introduces the concept of resource-aware modulo scheduling, and evaluates multi-criteria optimisation techniques to efficiently compute the set of all Pareto-optimal trade-off solutions.

CHAPTER 6 combines insights from all the previous chapters to tackle an integrated scheduling and allocation problem covering an entire kernel comprised of multiple loops and functions.

CHAPTER 7 summarises the contributions of the thesis, and outlines future research directions.

In this chapter, we establish the technical context for the contributions of this thesis. Readers unfamiliar with the basic HLS and modulo scheduling terminology find a brief introduction in Sections 2.1 to 2.3. The fundamentals of linear programming and integer linear programming are explained, for example, in Hamacher and Klamroth [37] and Wolsey [94]. From Section 2.4 on, we introduce the notation used throughout the thesis, formally define the Modulo Scheduling Problem and useful extensions, and survey prior modulo scheduling approaches.

2.1 COMMON COMPONENTS OF HIGH-LEVEL SYNTHESIS FLOWS

In this section, we will briefly introduce the ingredients that comprise a typical HLS flow. Refer to De Micheli [18] for a more detailed discussion of the fundamental concepts in architectural synthesis, and to Nane et al. [61] for a survey of HLS tools.

2.1.1 Frontend

The frontend is responsible for reading the input specification, and constructing an Intermediate Representation (IR) suitable for hardware generation.

As most HLS flows accept languages (or dialects thereof) originating from a software context, standard compiler technology is employed in this step. For example, Vivado HLS [91], LegUp [8] and Nymble [42] use LLVM [53], whereas Bambu [71] is based on GCC [33]. The prevalent IR used in HLS tools is the Control-Data-Flow Graph (CDFG) [35].

Consider the C code snippet in Listing 2.1, which does not compute anything useful, but helps us to illustrate the key concepts. The CDFG representation of the loop is shown in Figure 2.1. The nodes, which we call *operations*, represent the expressions and constants in the source code. The solid edges model the flow of values between the operations. There is one solid red edge that indicates a value flow across iteration boundaries. Here, this is only the case for the loop counter *i*, which is represented by the special ϕ -function customary in Static Single Assignment (SSA)-IRs [16]. The array *arr* in the source code will be mapped to shared memory, and accessed by two load as well as one store operation. In order to achieve the same order of the memory accesses as in the software program, the frontend runs a

The code fills an array with a Fibonacci-like sequence of numbers, but instead of writing back the sum of the two preceding cells, it uses a code sequence from [93] to compute the number of 1-bits in the sum.

Listing 2.1: An example loop

```

...
unsigned arr[N]; arr[0] = 0; arr[1] = 1;
loop1:
for (unsigned i = 2; i < N; ++i)
    unsigned a = arr[i-2];
    unsigned b = arr[i-1];
    unsigned x = (a+b) & 0xFF;
    x = ((x * 0x08040201) >> 3) & 0x11111111;
    x = ((x * 0x11111111) >> 28);
    arr[i] = x;
}
...

```

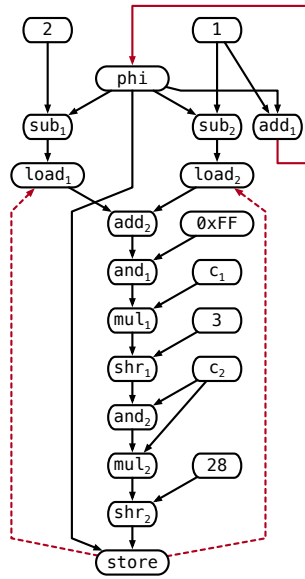


Figure 2.1: CDFG for code snippet in Listing 2.1. Let $c_1 = 0x08040201$, and $c_2 = 0x11111111$. Red edges denote inter-iteration dependencies. Dotted edges represent memory dependencies.

dependence analysis [34], and introduces additional edges (dashed in the example graph) that sequentialise memory accesses where needed. In our example, only inter-iteration edges are required, because the dependence analysis can prove that within one iteration, no conflicting memory accesses will occur. However, the analysis determines that the store operation writes to a memory address in iteration i , which is read in iterations $i + 1$ and $i + 2$. It follows that the store operations must always precede the next iterations' load operations.

It is possible and common to represent loop bodies with branches as a CDFG. As a preparation, *if-conversion* [59], a standard compiler transformation, is invoked. An operation that is only executed conditionally, in a branch controlled by a condition p , is afterwards associated with p as its *predicate*. In the CDFG representation, this will result in an additional edge from the predicate to the operation, indicating that its execution depends on the evaluation result of p .

2.1.2 Operator Library

An HLS environment provides a library of operator implementations, tailored to each supported target FPGA device. Typical HLS operator types perform the usual arithmetic and comparison functions, both for integer and floating-point, and memory/bus accesses. Additionally, extended math libraries, e.g. providing trigonometric operators, are common.

Operator types are characterised by the following metrics.

- The *latency* denotes after how many time steps the operator's result is available.
- The *blocking time* indicates after how many time steps an operator can accept new inputs. In case an operator is pipelined internally, its blocking time is less than the latency. Operators that can accept new input in every time step are called *fully-pipelined*, and are often available in HLS operator libraries.
- The *delay*, or more precisely, an incoming and an outgoing delay, specified in a physical amount of time, e.g. nanoseconds, which approximates the propagation time to the first, and from the last register inside the operator.
- The *resource demand*, which in the case of FPGAs is an amount of the low-level resources.

In general, the operator library contains variants for multiple combinations of input and output bitwidths, and may provide several implementations for the same functionality, but with different characteristics to choose from.

The latency and the blocking time are design choices made during the implementation of the operator template. In contrast, the delay

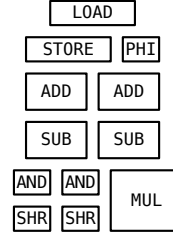


Figure 2.2: An operator allocation for the CDFG in Figure 2.1. The rectangle sizes hint at different resource demands for the operator types.

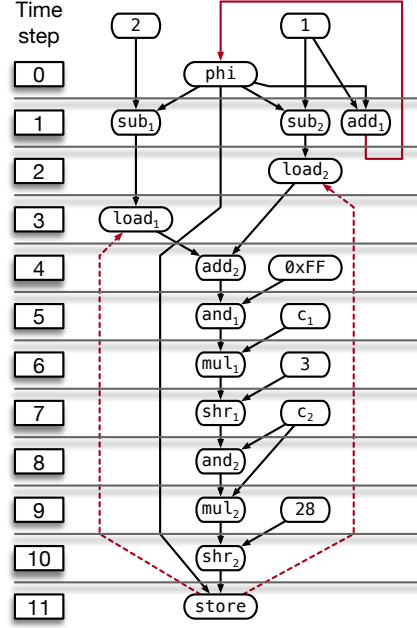


Figure 2.3: A schedule for the CDFG in Figure 2.1

and the resource demand are either obtained from an out-of-context logic synthesis run, or by using a model derived from such runs [61].

2.1.3 Algorithmic Steps

The core algorithmic steps in every HLS flow are *allocation*, *scheduling*, and *binding*.

In the allocation step, the number of instances, and an implementation variant (if available), is determined for each operator type. In the scheduling step, each operation is assigned to a time step. In the binding step, each operation is bound to an operator instance. Figures 2.2 to 2.4 illustrate these steps for the example CDFG in Figure 2.1.

While the practical consensus is to perform the three steps one after another, in the order allocation – scheduling – binding, they are all intertwined and form a classical phase-ordering problem [35].

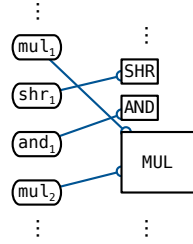


Figure 2.4: A binding, shown for a part of the CDFG in Figure 2.1

2.1.4 Controller Generation

HLS tools automatically generate a controller circuit that discretises the execution of the datapath into time steps, i.e. it starts the operations according to the computed schedule. Operations in different time steps are decoupled by registers that hold intermediate results. The controller can be as simple as a shift register, though more advanced implementations exist [41].

2.1.5 Backend

The microarchitecture comprised of datapath and controller is now ready to be emitted in an HDL, and could be handed off to an appropriate logic synthesis tool chain for the target device, which in the end will produce the bitstream required to set the configuration bits for all on-chip resources and the interconnect.

Note that HLS tools usually generate only a simple interface for data input and output. In consequence, nowadays the generated module is integrated into a larger system, such as an OpenCL [47] runtime or the TaPaSCo accelerator template [50], in order to obtain a complete design that handles the communication with the outside world.

We use the term “time step” in this thesis, as it describes the abstract concept, and there is not always a 1:1 correspondence between logical time steps and actual clock cycles of synchronous circuits.

2.2 KEY HIGH-LEVEL SYNTHESIS TECHNIQUES

We now introduce three key techniques in HLS that are relevant to the scheduling approaches presented in this thesis.

2.2.1 Operator Chaining

The frequency of the generated circuit is a major factor for the generated datapath’s performance. Typically, an HLS tool maintains a desired target cycle time Z for the datapath, which limits the time that can be spent in a single time step, and, at least theoretically, enables an operating frequency of $1/Z$.

In the example schedule in Figure 2.3, only independent operations share one time step, while dependent operations are assigned to

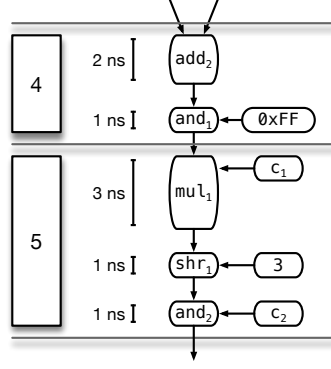


Figure 2.5: Application of operator chaining to the CDFG in Figure 2.1

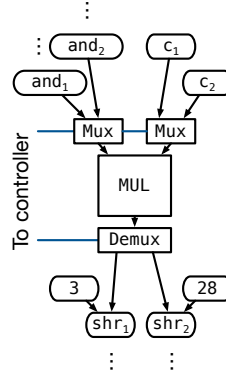


Figure 2.6: A multiplication operator, shared between two operations

individual time steps. However, not all operators require the same amount of time, e.g. logical function and constant bit shifts are trivial to perform on an FPGA, and have propagation delays $\ll Z$.

It follows that *chains* of dependent operations can be scheduled to the same time step, as long as the accumulated delay does not exceed the desired cycle time. In Figure 2.5, we annotate exemplary delays to a subset of the operations in Figure 2.1, and show the effect of chaining with a target cycle time of $Z = 5$ ns.

2.2.2 Operator Sharing

The highest possible performance for a given CDFG is achieved by the *fully spatial* microarchitecture, in which an individual operator is allocated for each operation. Most operator types are simple to implement on FPGAs, and have resource demands that are negligible compared to the amount of resources provided by modern devices. Therefore, it is reasonable for an HLS tool to treat them as practically *unlimited*, i.e. instantiate as many operators as needed.

For operator types with non-negligible resource demands, e.g. floating-point operators, the HLS tool may *share* operator instances across

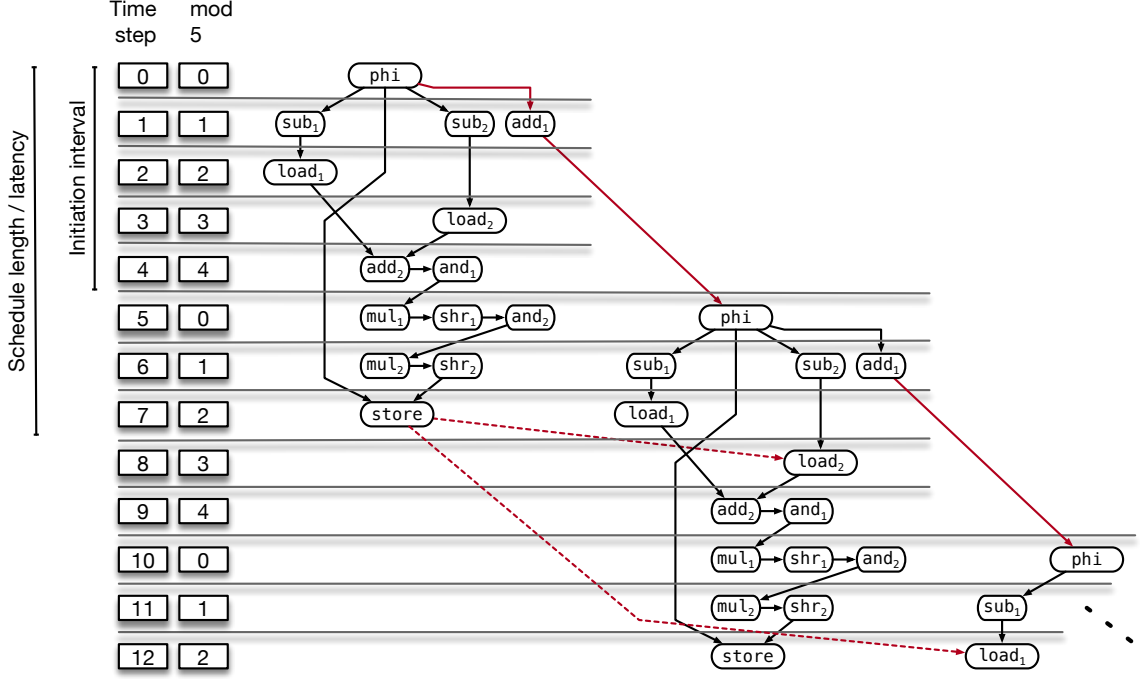


Figure 2.7: Loop pipelining with $II=5$, for the CDFG in Figure 2.1, with operator chaining. The CDFG constants have been omitted for clarity.

several operations, in order to trade a lower overall resource demand for potentially worse performance. In the generated datapath, these operators will then be time-multiplexed. Figure 2.6 illustrates the basic technique: multiplexers route different pairs of inputs to the operator. The result is routed by a demultiplexer to one of the consumers in the datapath. The multiplexers and the demultiplexer are switched by the generated controller circuit. Refer to [55] for a more advanced operator sharing implementation.

Note that the additionally needed (de-)multiplexers also occupy FPGA resources on their own. To that end, HLS tools usually only consider operator types for sharing if their resource demands significantly outweigh the costs for the multiplexing logic.

2.2.3 Loop Pipelining

As mentioned earlier, HLS tools must exploit all available sources of parallelism in order to achieve a meaningful speed-up compared to the execution on a software-programmable processor having a higher clock rate. This is especially true for the synthesis from sequential languages such as C.

One such source of parallelism is *loop pipelining*: the partially overlapping execution of subsequent loop iterations intended to increase the accelerator's throughput and the utilisation of the operators within

Schedulers that treat the Π as a rational number have been proposed [31, 81], but require extensions to the controller generation that are not available in current academic and commercial HLS tools.

the datapath. To this end, new loop iterations are started after a fixed number of time steps, called the *Initiation Interval* (Π).

Figure 2.7 shows the pipelined execution of the example CDFG from Figure 2.1. We assume the presence of one read and one write port to the shared memory where the array `arr` is stored. In consequence, the load operations cannot be scheduled to the same time step.

In this example, new iterations are initiated every 5 time steps, so we have $\Pi = 5$. A lower Π is not possible here, as otherwise the inter-iteration dependence between the store operation in the first iteration to the right load operation would be violated. Note that the left load operation depends on the value written *two* iterations earlier, and thus can overlap with the store operation in the directly preceding iteration.

In general, the following considerations highlight the possible benefits of loop pipelining. Let T be the latency of the datapath representing the loop body. Executing c iterations of the loop without overlap then takes $c \cdot T$ time steps. Assuming that the loop's inter-iteration dependences allow it to be executed with an initiation interval $\Pi < T$, then executing c iterations will require only $(c - 1) \cdot \Pi + T$ time steps, i.e. the last iteration is issued after $(c - 1) \cdot \Pi$ time steps and ends after the T time steps to fully evaluate the result of the datapath. This means that the smaller the interval is relative to the latency of the datapath, the higher is the theoretical speed-up achievable through loop pipelining.

Loop pipelining limits the amount of operator sharing that is possible, and vice versa.

2.3 MODULO SCHEDULING

The most practically relevant class of approaches to find a schedule suitable for loop pipelining is *modulo scheduling*. The name was coined by Rau [74], who together with Lam [52] laid most of the groundwork of the modulo scheduling framework, and refers to the following situation: Assume the allocation step determined that a_q instances of a shared operator type q should be available for the evaluation of a loop body. In a non-overlapping execution of the loop iterations, we would need to schedule the operations so that no more than a_q operators are used in any time step. In contrast, a pipelined execution of the iterations means that operations from different iterations are started in the same time step (cf. Figure 2.7). It follows that the aforementioned constraints with regards to the operator usage now have to consider congruence classes of time steps, *modulo* the Π .

*“Resources” vs.
“Operators”*

In the literature, and according to the established scheduling terminology, modulo scheduling is classified as a *resource*-constrained, cyclic scheduling problem. However, in the HLS-for-FPGA context, the term “resource” usually corresponds to the low-level device resources such as LUTs and FFs. The operations to be scheduled do not directly

Table 2.1: Problem signature of the Modulo Scheduling Problem

Symbol	Description
INPUT	
Q	$= Q^{\text{Sh}} \cup Q^{\infty}$ Operator types
Q^{Sh}	Shared operator types
Q^{∞}	Unlimited operator types
a_q	$\in \mathbb{N}, \forall q \in Q^{\text{Sh}}$ Allocation (i.e. number of operator instances) for shared operator type q
b_q	$\in \mathbb{N}, \forall q \in Q^{\text{Sh}}$ Blocking time (number of time steps) of shared operator q
g	$= (O, E)$ Dependence graph
O	$= \bigcup_{q \in Q} O^q$ Operations
O^q	$\forall q \in Q$ Operations using a specific operator type q
l_i	$\in \mathbb{N}_0, \forall i \in O$ Latency (number of time steps) of operation i
E	$\subseteq O \times O$ Dependence edges
d_{ij}	$\in \mathbb{N}_0, \forall (i \rightarrow j) \in E$ Edge distance (in number of iterations)
L_{ij}	$\in \mathbb{N}_0, \forall (i \rightarrow j) \in E$ Additional edge latency (number of time steps)
OUTPUT: A solution S, comprised of	
Π^S	$\in \mathbb{N}$ Initiation interval
t_i^S	$\in \mathbb{N}_0, \forall i \in O$ Schedule, i.e. a start time for each operation i

use these resources, but rather require access to a (potentially) limited number of *operators*, to fulfil their intended function. To that end, the terminology in this thesis hinges around the term “operator”.

2.4 FORMAL DEFINITION OF THE MODULO SCHEDULING PROBLEM

We now introduce the signature of the basic Modulo Scheduling Problem (**MSP**) used throughout this thesis. Tables 2.1 and 2.2 summarise the notation.

The input is comprised of two main parts, i.e. a specification of the *operator model* and the *dependence graph* representing the computation in one iteration of a pipelined loop. As a *solution* to the problem, we seek a feasible Initiation Interval (**II**) and a corresponding schedule. This

Table 2.2: Supplementary notations

Symbol	Description
$O^{\text{Sh}} = \bigcup_{q \in Q^{\text{Sh}}} O^q$	Operations using a shared operator type
$O^{\infty} = \bigcup_{q \in Q^{\infty}} O^q$	Operations using an unlimited operator type
$\sigma(i) \quad \sigma : O \rightarrow Q$ $\sigma(i) = q \Leftrightarrow i \in O^q$	Operator type used by operation i
$l_{ij} = l_i + L_{ij}, \forall (i \rightarrow j) \in E$	Required number of time steps between operations i and j
$T^S = \max_{i \in O} t_i^S + l_i$	Schedule length, i.e. latest finish time of all operations

In Chapter 5, we extend the problem to encompass scheduling and allocation

definition assumes that the three HLS phases allocation – scheduling – binding are performed separately and in that order. In consequence, the operator allocation is a parameter to the MSP, and a valid binding needs to be computed after scheduling.

OPERATOR MODEL The set of operator types Q is partitioned into the sets of shared types Q^{Sh} and unlimited types Q^{∞} . The HLS tool allocated a_q instances of each shared operator type $q \in Q^{\text{Sh}}$. q is characterised by its blocking time b_q , which means its instances can accept new input values from a different operation every b_q time steps.

This operator model corresponds to the “block reservation tables” described by Rau [74], as a single operator is used for b_q consecutive time steps, relative to the operation’s start time. Eichenberger [24] showed that in this case, a valid binding is guaranteed to exist if the number of users of an operator type does not exceed its allocation at any time. A small caveat here is that unrolling the schedule a certain number of times during hardware generation may be required if non-fully-pipelined operators are present (cf. Figure 2.10).

DEPENDENCE GRAPH The dependence graph $g = (O, E)$ is defined by its sets of operations and edges. The set of operations O is partitioned into the sets O^q that group together all operations using a specific operator type $q \in Q$. This means that each operation i is associated with exactly one operator type $\sigma(i)$, as specified by the mapping function σ . For notational convenience, we also define O^{Sh} to contain operations using any shared operator type, and O^{∞} to include operations using an unlimited operator type.

Each operation i is characterised by its latency l_i , i.e. the number of time steps after which the operation’s result is available. It is common,

but not required that all $i \in O^q$ derive the same latency from their operator type.

The dependence edges E model the data flow, and other precedence relationships between the operations. Each edge $(i \rightarrow j)$ has a distance d_{ij} , which indicates how many iterations later the dependence has to be satisfied. Intra-iteration dependences have $d_{ij} = 0$ and thus require that i precedes j in the same iteration. Conversely, edges with $d_{ij} \geq 1$ model inter-iteration dependences, which we also call *backedges* for short, as they often point in the opposite direction of the normal data flow. The dependence graph may contain cycles that contain at least one backedge, but not every backedge must span a cycle.

An edge may also carry a latency L_{ij} . We introduce $l_{ij} = l_i + L_{ij}$ as a shorthand notation for the smallest number of time steps that j can start after i , according to the edge $(i \rightarrow j)$.

Table 2.3 specifies the MSP instance corresponding to our running example from Figure 2.1.

SOLUTION A solution S is comprised of an integer initiation interval Π^S and the schedule, i.e. a start time t_i^S for each operation $i \in O$. The schedule length $T^S = \max_{i \in O} t_i^S + l_i$ is defined as the latest finish time of all operations, and corresponds to the latency of one loop iteration.

The solution is *feasible* if and only if the constraints (2.1) and (2.2) are satisfied.

The precedence constraints (2.1) ensure that enough time steps separate the start times of the endpoints of a dependence edge $(i \rightarrow j)$. For backedges, the right-hand side of the constraint represents j 's start time, only d_{ij} iterations later, as a new iteration will be initiated every Π^S time steps.

$$t_i^S + l_{ij} \leq t_j^S + d_{ij} \cdot \Pi^S \quad \forall (i \rightarrow j) \in E \quad (2.1)$$

The operator constraints (2.2) guarantee that no shared operator type q is oversubscribed in any congruence class modulo Π^S . Recall that per our operator model, an operation $i \in O^q$ exclusively uses a q -instance in its start time step t_i^S and the following $b_i - 1$ time steps. Formally, we enumerate these time steps with the help of β , and determine the congruence classes in which i uses its operator. Then, we simply count the number of operations that require a q -instance in every congruence class x .

$$\left| \{i \in O^q : x \in \{(t_i^S + \beta) \bmod \Pi^S : \forall \beta \in [0, b_i - 1]\}\} \right| \leq a_q \quad \forall x \in [0, \Pi^S - 1], \forall q \in Q^{\text{Sh}} \quad (2.2)$$

We will use the edge latency to statically limit the amount of chaining (Section 2.2.1), and in our problem-compression algorithm (Chapter 4).

W.l.o.g., we assume that the schedule starts at time step 0.

Table 2.3: MSP instance corresponding to the CDFG in Figure 2.1. Constants were omitted for brevity. A valid schedule is shown in Figure 2.7.

Q^∞	=	$\{\text{PHI}, \text{ADD}, \text{SUB}, \text{AND}, \text{SHR}\}$
Q^{Sh}	=	$\{\text{MUL}, \text{LOAD}, \text{STORE}\}$
Q	=	$Q^\infty \cup Q^{\text{Sh}}$
O^{PHI}	=	$\{\text{phi}\}$
O^{ADD}	=	$\{\text{add}_1, \text{add}_2\}$
O^{SUB}	=	$\{\text{sub}_1, \text{sub}_2\}$
O^{AND}	=	$\{\text{and}_1, \text{and}_2\}$
O^{SHR}	=	$\{\text{shr}_1, \text{shr}_2\}$
O^{MUL}	=	$\{\text{mul}_1, \text{mul}_2\}$
O^{LOAD}	=	$\{\text{load}_1, \text{load}_2\}$
O^{STORE}	=	$\{\text{store}\}$
O	=	$\bigcup_{q \in Q} O^q$
E	=	$\{(\text{phi} \rightarrow \text{sub}_1), (\text{phi} \rightarrow \text{sub}_2), (\text{phi} \rightarrow \text{add}_1),$ $(\text{sub}_1 \rightarrow \text{load}_1), (\text{sub}_2 \rightarrow \text{load}_2),$ $(\text{load}_1 \rightarrow \text{add}_2), (\text{load}_2 \rightarrow \text{add}_2),$ $(\text{add}_2 \rightarrow \text{and}_1), (\text{and}_1 \rightarrow \text{mul}_1), (\text{mul}_1 \rightarrow \text{shr}_1)$ $(\text{shr}_1 \rightarrow \text{and}_2), (\text{and}_2 \rightarrow \text{mul}_2), (\text{mul}_2 \rightarrow \text{shr}_2)$ $(\text{shr}_2 \rightarrow \text{store}), (\text{add}_1 \rightarrow \text{phi})$ $(\text{store} \rightarrow \text{load}_1), (\text{store} \rightarrow \text{load}_2)\}$
b_q	=	$1 \quad \forall q \in Q^{\text{Sh}}$
a_q	=	$1 \quad \forall q \in Q^{\text{Sh}}$
l_i	=	$1 \quad \text{for } i \in \{\text{phi}, \text{load}_1, \text{load}_2, \text{store}\}$
l_i	=	$0 \quad \text{for all other operations } i \in O$
$d_{\text{add}_1 \text{ phi}}$	=	1
$d_{\text{store load}_1}$	=	2
$d_{\text{store load}_2}$	=	1
d_{ij}	=	$0 \quad \text{for all other edges } (i \rightarrow j) \in E$
L_{ij}	=	$0 \quad \forall (i \rightarrow j) \in E$

Table 2.4: Chaining extension for the Modulo Scheduling Problem

Symbol	Description
INPUT	
$z_i^{\text{in}}, z_i^{\text{out}} \in \mathbb{R}^+, \forall i \in O$	Incoming and outgoing physical delay (e.g. in nanoseconds) of operation i
$Z \in \mathbb{R}^+$	Desired cycle time
OUTPUT A solution S , augmented with	
$z_i^S \in \mathbb{R}^+, \forall i \in O$	Physical start time of operation i within time step t_i^S

OBJECTIVE Modulo scheduling is usually a bi-criteria optimisation problem, with the minimisation of the II being the first objective. In this thesis, if not stated otherwise, we minimise the schedule length as the second objective. Note that in practice, minimising the II is far more important than optimising for any additional objective. To that end, we will attempt to minimise the tuple (II^S, T^S) lexicographically.

We discuss other possible objectives in Section 2.7.1.

2.5 SCHEDULER SUPPORT FOR OPERATOR CHAINING

The MSP is defined in terms of abstract time steps, which later correspond to clock cycles in the design generated by the HLS tool. An adjacent concern for HLS-schedulers is to limit the amount of chaining that results from the computed schedule, in order to meet a desired cycle time Z for the generated design (cf. Section 2.2.1).

Table 2.4 summarises a suitable extension to the MSP definition. Each operation i incurs an incoming and an outgoing physical propagation delay (z_i^{in} , respectively z_i^{out}). As part of a solution S , we determine a physical start time z_i^S , relative to the beginning of the cycle, for each operation i . Intuitively, we need to ensure that no chain of data-dependent operations scheduled to the same time step exceeds the target cycle time. More precisely, in (2.3) we recursively define the physical start time of an operation j as the latest physical finish time of j 's predecessors regarding the dependence edges. Given an edge $(i \rightarrow j)$, a chain is formed (or continued) only if operation i finishes in the same time step as j starts. Operations amenable to chaining often have a latency of zero time steps. In this case, there is effectively only one propagation delay, and we expect the specified incoming and outgoing delays to be equal. The cycle time constraints (2.4)

express that an operation j must start early enough in its time step to accommodate its incoming delay.

$$z_j^S = \max \{z_i^S + z_i^{\text{out}} \mid \forall (i \rightarrow j) \in E : t_i^S + l_i = t_j^S\} \cup \{0\} \quad \forall j \in O \quad (2.3)$$

$$z_j^S + z_j^{\text{in}} \leq Z \quad \forall j \in O \quad (2.4)$$

The basic idea to enforce constraints (2.4) is to introduce additional time steps between operations that would exceed the cycle time when chained together [15]. For that purpose, we add extra edges ($u \rightarrow v$) with a latency $L_{uv} = 1$ to the MSP instance, so that the cycle time constraints can be transparently considered in the remaining parts of the HLS flow, e.g. to compute more precise bounds prior to the actual scheduling.

Algorithm 1 Algorithm to compute chain-breaking edges

```

1: The algorithm operates on the acyclic subgraph
    $g' = (O, \{(i \rightarrow j) \in E : d_{ij} = 0\})$  of  $g$ .
2: for all operations  $v \in O$ , in topological order do
3:    $\text{CHL}_v[v] \leftarrow 0.0$ 
4:   for incoming edges  $(u \rightarrow v)$  do
5:     if  $L_{uv} > 0$  then
6:       continue
7:     if  $l_u > 0$  then
8:        $\text{CHL}_v[u] \leftarrow z_u^{\text{out}}$ 
9:       continue
10:    for all chain-starting operations  $p \in \text{CHL}_u$  do
11:       $\text{CHL}_v[p] \leftarrow \max(\text{CHL}_u[p] + z_u^{\text{out}}, \text{CHL}_v[p])$ 
12:    for all chain-starting operations  $p \in \text{CHL}_v$  do
13:      if  $\text{CHL}_v[p] + z_v^{\text{in}} > Z$  then
14:        introduce chain-breaking edge  $(p \rightarrow v)$  with  $L_{pv} = 1$ 
15:        remove  $p$  from  $\text{CHL}_v$ 

```

Algorithm 1 outlines a procedure to compute the chain-breaking edges for a given MSP instance. As preparation, we determine the acyclic subgraph of the dependence graph, and associate a dictionary CHL_v with each operation v . $\text{CHL}_v[p]$ denotes the accumulated delay on the longest chain from p to v .

We visit each operation v in topological order (Line 2), and register v in its own dictionary (Line 3). Then, we inspect v 's incoming edges (Line 4). Edges already carrying an extra latency do not propagate any delays (Line 5) and can be skipped. If the source operation u of the edge has a non-zero latency, none of u 's incoming chains are extended by it. We only register u 's outgoing delay (Line 8), and continue with the next incoming edge.

Otherwise, u is combinatorial. The loop starting in Line 10 propagates u 's incoming chains, suffixed by u itself, to v : in Line 11, we set

*We assume here that
no operator chaining
occurs along
backedges.*

Table 2.5: Chaining subproblem example, for the MSP in Table 2.3

Z	$=$	5 ns	
z_i^{in}	$=$	Z	for $i \in \{\text{phi}, \text{load}_1, \text{load}_2, \text{store}\}$
z_i^{out}	$=$	0 ns	for $i \in \{\text{phi}, \text{load}_1, \text{load}_2, \text{store}\}$
$z_i^{\text{in}} = z_i^{\text{out}}$	$=$	3 ns	for $i \in \{\text{mul}_1, \text{mul}_2\}$
$z_i^{\text{in}} = z_i^{\text{out}}$	$=$	2 ns	for $i \in \{\text{add}_1, \text{add}_2\}$
$z_i^{\text{in}} = z_i^{\text{out}}$	$=$	1 ns	for $i \in \{\text{and}_1, \text{and}_2, \text{shr}_1, \text{shr}_2\}$

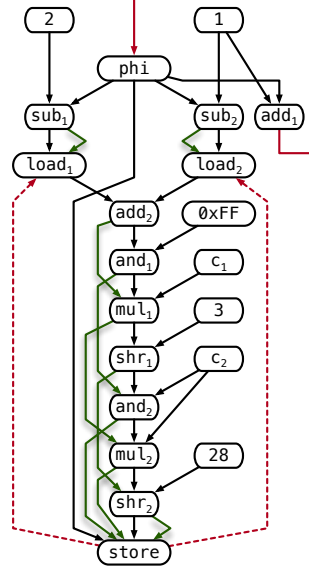


Figure 2.8: Additional chain-breaking edges (green, edge latency = 1) for the extended MSP defined by Table 2.3 and Table 2.5

$\text{CHL}_v[p]$ to the longest delay from p to u , plus u 's outgoing delay, if that sum is greater than any previous entry for $\text{CHL}_v[p]$ (e.g. set by another incoming edge).

After we have processed all incoming edges, we inspect all incoming delays registered in CHL_v in the loop in Line 12. If chaining v to a chain starting at p would exceed the desired cycle time, as checked in Line 13, we introduce a chain-breaking edge (Line 14), and remove p from v dictionary accordingly (Line 15).

Let us assume a desired cycle time of 5 ns, and physical delays as in Table 2.5 for the running example (Table 2.3). Then, applying the algorithm above yields the green chain-breaking edges in Figure 2.8.

For the remainder of this thesis, we will not address the chaining support explicitly, but instead assume that the MSP instance at hand has been extended to contain the chain-breaking edges as discussed in this section.

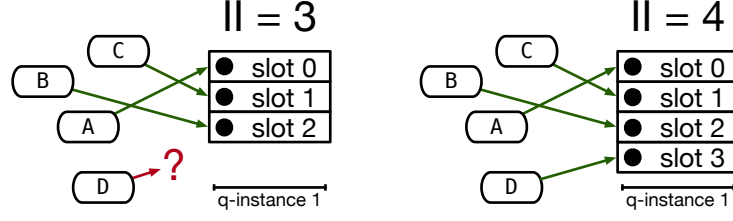


Figure 2.9: Application of the pigeonhole principle to determine a lower bound for the Π with regards to a shared operator type q with $b_q = 1$. Each q -instance provides Π -many slots to accept inputs from different operations. If only one q instance is available (as shown), the Π must therefore be at least 4 in order to accommodate operations A–D.

2.6 BOUNDS FOR THE Π SEARCH SPACE

Before scheduling, it is common to compute bounds from the MSP instance that restrict the search space for the smallest feasible Π to sensible values.

2.6.1 Lower Bound

The lower bound Π^{\min} is usually defined as in (2.5), i.e. as the maximum of the recurrence-constrained minimum Π and the operator-constrained minimum Π [74].

$$\Pi^{\min} = \max(\Pi^{\text{rec}}, \Pi^{\text{opr}}) \quad (2.5)$$

$$\Pi^{\text{rec}} = \max_{\mathcal{C} \in \text{cycles}(G)} \left\lceil \frac{\sum_{(i \rightarrow j) \in \mathcal{C}} l_{ij}}{\sum_{(i \rightarrow j) \in \mathcal{C}} d_{ij}} \right\rceil \quad (2.6)$$

$$\Pi^{\text{opr}} = \max_{q \in Q^{\text{Sh}}} \left\{ \max \left(\left\lceil \frac{|O^q| \cdot b_q}{a_q} \right\rceil, b_q \right) \right\} \quad (2.7)$$

RECURRENCE-CONSTRAINED MINIMUM Π The recurrences (cycles) in the dependence graph impose a lower bound for any feasible Π . The repeated application of the precedence constraint (2.1) to the edges in a recurrence leads to the form in (2.6). However, computing the lower bound in this way would require the enumeration of all cycles in the graph. Instead, we compute Π^{rec} as the optimal solution to the ILP (2.8)–(2.10), defined for integer variables t_i that model the start time step for each operation i , and an integer variable Π that models the recurrence-induced Π to be minimised.

As noted by Dinechin [19], this is a resource-free cyclic scheduling problem, which can be solved optimally in polynomial time.

$$\min \Pi \quad (2.8)$$

$$\text{s.t. } t_i + l_{ij} \leq t_j + d_{ij} \cdot \Pi \quad \forall (i \rightarrow j) \in E \quad (2.9)$$

$$\Pi \in \mathbb{N}, t_i \in \mathbb{N}_0 \quad \forall i \in O \quad (2.10)$$

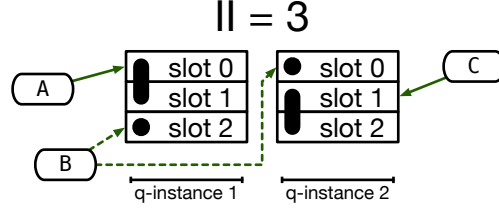


Figure 2.10: Now considering a shared operator type q with $b_q = 2$. Two q -instances together provide six slots, which is just enough to accommodate three operations blocking two slots each. A valid binding does exist, but will require unrolling the finished schedule [24].

OPERATOR-CONSTRAINED MINIMUM II The definition (2.7) of Π^{opr} follows from the operator constraints (2.2). Considering a shared operator type q , every q -instance provides Π -many *slots* to accept inputs from a different operation. These slots correspond to the congruence classes modulo Π of the operations' start times. We have $|O^q|$ operations that each block their associated operator instance for b_q time steps. It follows that a feasible Π must respect the inequality (2.11).

$$\underbrace{\Pi \cdot a_q}_{\text{available slots}} \geq \underbrace{|O^q| \cdot b_q}_{\text{required slots}} \quad (2.11)$$

Intuitively, this lower bound is an application of the pigeonhole principle, as illustrated in Figure 2.9. Figure 2.10 shows the situation for an operator type with a longer blocking time, and outlines the aforementioned problem of finding a binding after scheduling.

Note that an operation can never be interrupted and continued on an additional operator instance. To that end, the Π must be greater or equal to the maximum blocking time across all shared operator types.

Shared operator types in the HLS context usually are fully-pipelined, so this is only a minor concern.

Consider the running example that has been augmented with chain-breaking edges in Figure 2.8. Here, we have $\Pi^{\text{rec}} = 5$, set by the recurrence spanned by the backedge ($\text{store} \rightarrow \text{load}_2$). Both the **LOAD** and **MUL** operator types have two users. As we allocated only one instance of each, the Π^{opr} is 2. All shared operator types are fully-pipelined. Altogether, we compute $\Pi^{\text{min}} = 5$ for the example.

2.6.2 Upper Bound

A trivial upper bound Π^{max} is the length of any operator-constrained *non-modulo* schedule. IIs larger than this value indicate that it would actually be faster to execute this loop in a non-overlapping manner. We use a non-modulo SDC scheduler [15] with heuristic operator constraints to quickly compute such a fallback schedule, and use its length to define Π^{max} .

2.7 SURVEY OF MODULO SCHEDULING APPROACHES

This section highlights common themes in modulo schedulers, and categorises the landscape of approaches previously presented in the literature.

2.7.1 Target Architecture

Loop pipelining is most useful when the target architecture provides multiple parallel functional units. Software-programmable out-of-order processors rely on dynamic scheduling to derive the appropriate execution sequences at run-time. Parallel in-order processors (such as **VLIW** architectures), or statically scheduled hardware accelerators created by HLS, however, rely on the compiler/synthesis tool to pre-compute their execution schedules, almost always by using a modulo scheduler.

In both application areas, the basic MSP is the same, with the primary objective being to find a schedule with the minimally feasible II. Nevertheless, instances from a VLIW context differ from HLS instances in several aspects, and in consequence, so do the schedulers proposed to solve them.

RESOURCE/OPERATOR MODELS VLIW processors provide *resources* such as arithmetic logic units, register files, and communication busses, which are fixed in the processor’s microarchitecture. In consequence, VLIW modulo schedulers typically face tight resource constraints (2.2) in each time step and for all operations. Additionally, depending on the complexity of the microarchitecture, operations may require access to several resources in different time steps relative to their start time, encoded in form of reservation tables [25, 74] that have to be considered during scheduling to detect potential resource conflicts. Examples for schedulers proposed with VLIW architectures in mind are [20, 26, 40, 52, 56, 74].

In contrast, an FPGA is comprised of a fixed number of low-level resources, which are not bound directly to operations, but rather used to instantiate *operators* that perform the intended function. Typically, operations are associated with only one operator type, resulting in a much simpler operator model in which it suffices to specify how many time steps an operator is blocked by an operation, relative to its start time.

HLS operators also exhibit properties that are not possible in a VLIW context (cf. Sections 2.2.1 to 2.2.2). First, many operator types have simple FPGA implementations, allowing the HLS tool to instantiate them as needed. Operations using these operator types therefore are not “resource-constrained” in the classical sense in the scheduling problem. Secondly, operators may have a latency of zero time steps,

and only incur a small physical delay. Therefore, modulo schedulers targeting HLS (e.g. [7, 17, 95]) may take advantage of the simpler operator model, but need to support unlimited and chainable operators efficiently.

BINDING A modulo scheduler must model the binding between the operations and resources/operators either if it should be optimised according to a secondary objective (see next paragraph), or if it is decisive for the feasibility of the computed schedule. As mentioned in Section 2.4, for HLS operator types it is sufficient to ensure that the allocated operator instances are not oversubscribed. In contrast, VLIW modulo schedulers may need to detect more elaborate resource conflicts [25], depending on the complexity of the processor’s microarchitecture, and thus are required to consider the binding during scheduling, typically in form of a Modulo Reservation Table (MRT) [52]. An MRT is a table with rows for the available resource/operator instances, and columns for the congruence classes (cf. Figure 3.1). At most one operation can be assigned to each table cell.

SECONDARY OBJECTIVES The choice of a secondary optimisation objective, after finding the smallest feasible II, also largely depends on a scheduler’s target architecture.

In the simplest case, no secondary objective is imposed at all, which means that we are only interested in finding any feasible schedule for a determined II. Minimising the schedule length, which corresponds to the latency (or makespan) for the computation of one loop iteration, is common in the HLS context. VLIW modulo schedulers often attempt to achieve a schedule with a low register demand in the subsequent code generation step. The direct way to model this objective is to minimise the maximum number of intermediate values that are live in any time step [27]. As a simpler alternative, the accumulation of the lifetimes of such values can be minimised, as, for example, in [19, 40, 56]. Sittel et al. [79] used a similar objective to reduce the number of distinct connections in a HLS datapath.

The HLS domain, with its application-specific microarchitectures, allows for the resource demand of the data path to be an optimisation objective, which was explored by Fan et al. [29] and Šůcha and Hanzálek [86].

2.7.2 Search Strategies for the II

Almost all known modulo scheduling approaches try to find the smallest feasible II by attempting to schedule a given MSP instance with several *candidate* IIs, until a feasible solution is found. In the context of one scheduling attempt, the II is a constant parameter.

Normally, the search begins at the lower bound, Π^{\min} , and progresses to the next Π in case no feasible solution was found [74]. Binary search [76, 86] can be used alternatively, assuming that all candidate Π s after the first feasible one are also feasible, which Lam [52] notes may be false, but only for particularly complex VLIW resource models.

To the best of our knowledge, the only exception to the iterative Π minimisation approach is the constraint programming formulation by Bonfietti et al. [6].

2.7.3 *Heuristic vs. Exact Approaches*

Modulo schedules can be determined either by *heuristics* or by using mathematical formulations to solve the problem *exactly*. While heuristics are not guaranteed to find the optimal solution (and often only produce feasible solutions), they usually come with a shorter computation time compared to the exact approaches, which are typically associated with impractically long runtimes.

Intuitively, what makes modulo scheduling “hard” (even NP-hard in general [52]) is the interaction of the inter-iteration dependences with the resource/operator constraints. The former impose deadlines for the operations, while the latter may require operations to be scheduled later than their earliest possible start time due to resource/operator conflicts [74].

We categorise approaches according to the framework they are formulated within, as it is decisive for the scheduling time, and quality of results to be expected from a particular algorithm.

NO FRAMEWORK Most of the early heuristic approaches were standalone algorithms [40, 56, 74] that traversed the input graph one or more times, and determined the start time for the operations one after the other. These algorithms might get stuck at some point, if the partial schedule built so far cannot be extended for the next operation. A common theme to resolve this situation is the use of backtracking, i.e. the unscheduling of some or all operations before restarting the scheduling procedure. Codina, Llosa and González [11] explain the differences in the most relevant approaches, and provide a detailed performance evaluation.

SYSTEM OF DIFFERENCE CONSTRAINTS Linear programs, defined in terms of real decision variables, are in general not suitable for modelling scheduling problems, because the operations’ start time need to be integer. However, one important theoretical result is that a linear program has an optimal *integer* solution if its constraint matrix is totally unimodular [78].

This insight has been used by Dinechin [19] in a resource-free VLIW modulo scheduling approach. Furthermore, the property holds for Systems of Difference Constraints (SDCs), a subclass of linear programs in which all constraints take the form $x - y \leq c$ for decision variables x and y , and a constant integer value c [15].

SDCs are able to capture the kinds of timing constraints required in an HLS modulo scheduler, including support for operator chaining. However, there is no exact translation of the operator constraints into the SDC framework. To that end, several strategies to augment an SDC-based scheduler to handle the operator constraints have been investigated.

The approaches proposed by Zhang and Liu [95] and Canis, Brown and Anderson [7] both use an MRT internally, and share the following basic procedure. A scheduling attempt starts with the computation of a non-operator-constrained schedule. Then, the algorithms iteratively try to assign shared operations to their designated time steps. The MRT is queried for operator conflicts, and if no conflicts arise, an operation's assignment is fixed in the underlying SDC by adding new equality constraints. Otherwise, a new constraint to move the conflicting operation to the next time step is introduced. The SDC is solved afterwards to check the feasibility of the current partial schedule. Should the schedule become infeasible, the MODULOSDC algorithm [7] uses backtracking to revert some of the previous assignments and resumes. In contrast, Zhang and Liu [95] propose to prioritise operations in a way to minimise the perturbation of their assignment to the MRT, and give up on a scheduling attempt if an assignment results in an infeasible partial schedule.

The aforementioned approaches, as well as the work by Souza Rosa, Bouganis and Bonato [84] who combined an SDC with a genetic algorithm to evolve the assignment of operations to the MRT, have to be classified as heuristics, because the decoupling of the scheduling and MRT handling parts. Even if the SDC is minimised according to a linear objective, e.g. the schedule length, an optimal solution is only optimal for a particular, heuristically determined MRT assignment. In consequence, it is impossible to prove that an MSP is infeasible for a certain Π without enumerating all possible MRT assignments.

Recently, Dai and Zhang [17] proposed an exact SDC-based modulo scheduling approach that models conflicting usage of operators as a Boolean satisfiability (SAT) problem. The SDC and SAT part of the formulation are invoked in an alternating way, and improve the respective other model successively, until a feasible schedule is found, or infeasibility was proven. They report shorter scheduler run-times compared to a preliminary version [63] of the ILP formulation presented in Chapter 3, and attribute the speed-up to the advantageous scaling properties of SAT solvers in this setting.

INTEGER LINEAR PROGRAMS Previously, the MSP has also been modelled exactly as an ILP for a single candidate interval Π . Several formulations have been proposed that can be classified roughly according to their modelling of the operations' start times t_i , and handling of the resource/operator constraints.

Time-indexed formulations use binary decision variables t_i^x per operation i and time step x to represent that i starts in x . The resource usage is constrained by comparing the sum of the t_i^x variables over the relevant time steps with the number of available resource/operator instances.

The formulation by Dinechin [20] exclusively uses such binaries to model the time steps up to a predefined time horizon, i.e. an upper bound T^{\max} for the schedule length, as $t_i = \sum_{x=0}^{T^{\max}} x \cdot t_i^x$. As the formulation cannot be efficiently solved for large instances, a large neighbourhood search (LNS) heuristic is proposed. It speeds up the scheduling process as a schedule for $\Pi - 1$ can easily be constructed (if it exists) given a feasible solution for Π .

Zhao et al. [97] use a time-indexed formulation to implement a mapping-aware scheduler, which does not work with fixed, predetermined delays to support operator chaining, but considers the actual mapping of nets to look-up tables during scheduling. However, in their experiments, adding the technique to the ILP slowed the solver down significantly.

In the formulation by Eichenberger and Davidson [26], the start times are decomposed according to the Euclidean division by Π , as $t_i = y_i \cdot \Pi + \sum_{x=0}^{\Pi-1} x \cdot m_i^x$. Here, binary decision variables model the assignment of the start time to the congruence classes $0 \dots \Pi - 1$, but integer variables y_i represent the so-called stage, i.e. the multiple of Π contained in t_i . The formulation is a frequent point of reference in this thesis. To that end, we present the complete formulation, adapted to our notation, in Section 2.8.1.

Ayala and Artigues [4] compare the aforementioned time-indexed formulations and report that the one by Eichenberger and Davidson [26] is always faster on their set of industrial instances. The authors obtain stronger variants of the formulations by applying a Dantzig-Wolfe decomposition, but rely on a column-generation scheme to cope with the increased number of decision variables. Fan et al. [29] extend the formulation by Eichenberger and Davidson [26] to also compute a binding, a requirement for their cost-sensitive modulo scheduler.

Formulations using *integer* variables to directly model the (potentially decomposed) start times promise to work with fewer decision variables overall. A common element in integer-based formulations is the concept of *overlap* variables, i.e. binary variables that express an ordering relation on the values of the $t_i \bmod \Pi$. Such formulations were proposed by Altman, Govindarajan and Gao [2] and Šůcha and Hanzálek [86]. As the latter (and more recent) formulation is part of

our experimental evaluation in Chapter 5, we present it in our notation in Section 2.8.2.

OTHER EXACT APPROACHES In addition to ILPs, the MSP can be solved with an enumeration scheme and extensive pruning [1], or Constraint Programming (CP). The CP framework allows to formulate more powerful constraints, and therefore needs less abstractions as an ILP model, making an integrated handling of the Π minimisation possible, as shown by Bonfietti et al. [6]. Their approach achieves competitive or better performance compared to heuristics and ILP formulations by implementing highly problem-specific search strategies instead of the more generic branch-and-bound techniques typically employed in ILP solvers.

2.8 PRIOR ILP FORMULATIONS

The survey in the previous section also yielded the insight that no generally accepted notation for describing MSP instances exists. To make the features of the two most relevant works easier to reference in the following chapters, we present them here in our notation.

2.8.1 Formulation by Eichenberger and Davidson

Eichenberger and Davidson [26] presented the following ILP formulation, which we reference as EDFORM in the remainder of this thesis.

$$\min T \quad (2.12)$$

$$\text{s.t. } \sum_{x=0}^{\Pi-1} m_i^x = 1 \quad \forall i \in O \quad (2.13)$$

$$\sum_{i \in O^q} \sum_{x'=0}^{b_q-1} m_i^{(x-x') \bmod \Pi} \leq a_q \quad \forall x \in [0, \Pi-1], \forall q \in Q^{\text{Sh}} \quad (2.14)$$

$$\sum_{x'=x}^{\Pi-1} m_i^{x'} + \sum_{x'=0}^{(x+l_{ij}-1) \bmod \Pi} m_j^{x'} + y_i - y_j \leq d_{ij} - \left\lfloor \frac{x+l_{ij}-1}{\Pi} \right\rfloor + 1 \quad \forall x \in [0, \Pi-1], \forall (i \rightarrow j) \in E \quad (2.15)$$

$$l_i + y_i \cdot \Pi + \sum_{x=1}^{\Pi-1} x \cdot m_i^x \leq T \quad \forall i \in O \quad (2.16)$$

$$m_i^x \in \{0, 1\} \quad \forall x \in [0, \Pi-1], \forall i \in O \quad (2.17)$$

$$y_i \in \mathbb{N}_0 \quad \forall i \in O \quad (2.18)$$

$$T \in \mathbb{N}_0 \quad (2.19)$$

The formulation assumes the Π to be a parameter, and thus requires an external driver that supplies candidate Π s until a feasible solution is

This multiple is also called the “stage” of the operation.

found. Each operation i 's start time is decomposed into a multiple y_i of the candidate Π and Π -many binaries m_i^x that encode i 's congruence class. (2.17)–(2.18) are the corresponding domain constraints. After a solution S is found, the start time t_i^S for every operation $i \in O$ can be extracted as $t_i^S = y_i \cdot \Pi^S + \sum_{x=1}^{\Pi^S-1} x \cdot m_i^x$.

(2.14) count the operations that occupy an operator instance in each of the congruence classes, taking into account that an operation blocks a q -operator for b_q consecutive time steps. For example, let $\Pi = 5$ and $b_q = 2$. Then, i uses an instance of q in a particular congruence class x , say $x = 3$, if it starts in congruence classes $3 - 0 \bmod 5 = 3$ or $3 - 1 \bmod 5 = 2$.

The unique feature of the EDFORM are the 0-1-structured dependence constraints (2.15). “0-1-structured” means that every decision variable occurs only once, and is multiplied by either -1, 0, or 1. These constraints have the effect that the LP solutions of the ILP already are mostly integer (similar to the SDC framework, in which the LP solution is guaranteed to be integer). In consequence, significantly fewer branch-and-bound nodes need to be computed during the ILP solving process [26], and an almost tenfold speed-up over was reached over a prior version [28]. However, the drawback of this modelling is that each edge is represented by Π -many constraints.

To gain an intuition of how constraints (2.15) work in the simplest case (refer to [26] for a general, formal derivation), let us assume we have a candidate interval $\Pi = 3$, and an edge $(u \rightarrow v)$ with $l_{uv} = 1$ and $d_{uv} = 0$. This results in the constraints (2.20)–(2.22).

$$(m_u^0 + m_u^1 + m_u^2) + (m_v^0) + y_u - y_v \leq 1 \quad x = 0 \quad (2.20)$$

$$(m_u^1 + m_u^2) + (m_v^0 + m_v^1) + y_u - y_v \leq 1 \quad x = 1 \quad (2.21)$$

$$(m_u^2) + (m_v^0 + m_v^1 + m_v^2) + y_u - y_v \leq 1 \quad x = 2 \quad (2.22)$$

Note that $m_i^0 + m_i^1 + m_i^2 = 1$ for all operations i , due to (2.13).

Recall that the edge is supposed to ensure that operation u finishes before v starts. (2.20) expresses, “if v is scheduled to congruence class 0, then u must be scheduled an entire stage ahead of v .” (2.21) reads, “if u is scheduled to congruence classes 1–2, then v must be either scheduled to congruence class 2, or u must be scheduled an entire stage ahead of v .” Lastly, (2.22) means, “if u is scheduled to congruence class 2, then u must be scheduled an entire stage ahead of v .” Together, these three constraints establish the edge’s desired semantic.

We added the decision variable for the schedule length (2.19), constraints (2.16) for its definition, and use it to change the objective (2.12) to a schedule length minimisation. In our implementation, we only introduce the constraints for sink nodes in the graph, i.e. operations without outgoing zero-distance edges.

2.8.2 Formulation by Šůcha and Hanzálek

We now adapt the unit processing time formulation by Šůcha and Hanzálek [86] to our notation, which we will refer to as SHFORM. In their terminology, “unit processing time” means that all shared operator types are fully pipelined. They also present a variant of the formulation suitable for arbitrary blocking times.

The SHFORM requires that the candidate Π is an externally specified parameter.

$$\text{Let } \mathcal{D} = \left\{ (q, i, j) \mid \forall q \in Q^{\text{Sh}} : \forall i, j \in O^q, i < j \right\}$$

$$\min T \quad (2.23)$$

$$\text{s.t. } m_j + y_j \cdot \Pi - m_i - y_i \cdot \Pi \geq l_{ij} - \Pi \cdot d_{ij} \quad \forall (i \rightarrow j) \in E \quad (2.24)$$

$$m_i - m_j + \Pi \cdot \mu_{ij}^{(x)} + (1 - \Pi) \cdot \mu_{ij}^{(y)} \geq 1 \quad \forall (q, i, j) \in \mathcal{D} \quad (2.25)$$

$$m_i - m_j + \Pi \cdot \mu_{ij}^{(x)} - \mu_{ij}^{(y)} \leq \Pi - 1 \quad \forall (q, i, j) \in \mathcal{D} \quad (2.26)$$

$$-\mu_{ij}^{(x)} + \mu_{ij}^{(y)} \leq 0 \quad \forall (q, i, j) \in \mathcal{D} \quad (2.27)$$

$$\sum_{j \in O^q : i \neq j} \mu_{ij}^{(y)} \leq a_q - 1 \quad \forall i \in O^q, \forall q \in Q \quad (2.28)$$

$$y_i \cdot \Pi + m_i + l_i \leq T \quad \forall i \in O \quad (2.29)$$

$$y_i, m_i \in \mathbb{N}_0 \quad \forall i \in O \quad (2.30)$$

$$\mu_{ij}^{(x)}, \mu_{ij}^{(y)} \in \{0, 1\} \quad \forall (q, i, j) \in \mathcal{D} \quad (2.31)$$

$$T \in \mathbb{N}_0 \quad (2.32)$$

In the model, an operation i 's start time is decomposed into two integer variables (2.30), i.e. the multiple y_i of the Π , and the index of the congruence class m_i . Within a feasible solution S , the start time t_i^S is derived as $t_i^S = y_i \cdot \Pi^S + m_i$.

Constraints (2.24) model the MSP's precedence constraints (2.1), albeit in the aforementioned decomposed form.

Constraints (2.25)–(2.27) define the binary variables $\mu_{ij}^{(x)}$ and $\mu_{ij}^{(y)}$ (2.31) to the semantic shown in (2.33). These definitions are introduced for unique pairs of operations using the same shared operator type. We assume the existence of an arbitrary total order “<” among the operations, and use the symbol \mathcal{D} as a short-hand notation for the constraints' domain specification.

$$\mu_{ij}^{(x)} = \begin{cases} 1 & m_i \leq m_j \\ 0 & \text{otherwise} \end{cases} \quad \mu_{ij}^{(y)} = \begin{cases} 1 & m_i = m_j \\ 0 & \text{otherwise} \end{cases} \quad (2.33)$$

If $\mu_{ij}^{(y)} = 1$, i and j cannot share the same operator instance. Constraints (2.28) count these conflicts per operation, and ensure that enough operator instances are available for the operation itself, plus all others it is in conflict with. Note though that we use the weaker form of

their constraints (9 in [86]) here, i.e. before applying their Lemma 1, as otherwise the number of constraints in the ILP would be dependent on the allocation, which in consequence would prevent the resource-aware extension in Chapter 5.

We add a decision variable for the schedule length (2.32), define it with constraints (2.29), and request to minimise it (2.23). Again, these constraints are only required for the graph’s sink nodes in an implementation of the scheduler.

2.9 RESEARCH GAP

We identify the three main research gaps.

1. There is lack a of comprehensive studies of exact modulo schedulers in the HLS context, arguably also because there is no representative data set available for comparison.

The SHFORM is intended to be used in HLS, but evaluated in [86] with only a handful of instances. All other recent exact approaches [4, 6, 20] were tested with a data set obtained from a VLIW software compiler. The available literature therefore does not address the question if exact modulo scheduling can be practical in HLS, and how well approaches previously proposed for a VLIW context work within an HLS environment.

2. ILPs appear to be the most popular framework for exact modulo scheduling, but the iterative search for the optimal Π apparently has been accepted as compulsory.

We acknowledge the effort by Bonfietti et al. [6] here, but note that such an approach involves a significant amount of engineering work using a CP solver library, whereas an ILP model is more flexible and can be passed almost verbatim to the ILP solver’s API, where it may benefit from the continuous improvements to the solver’s internal algorithms without modification.

3. In HLS for FPGA, the amount of loop pipelining should be subject to the available *resources*, not pre-allocated operators. We have two equally important, but competing objectives, which should be subject to multi-criteria optimisation.

As mentioned earlier, Fan et al. [29] consider the resource demand in their approach, but fix the number of functional units before scheduling. Šůcha and Hanzálek [86] lay the groundwork for a variable operator allocation, but do not investigate the exploration of different trade-off solutions.

We address the first aspect in Chapters 3 to 4. The HLS-specific data set was made public as part of the HATSCHEt scheduling toolkit [80].

The ILP-based MOOVAC formulation, also presented in Chapter 3, refutes the second point. It enables a novel strategy to tackle the

Note that [17] was published in 2019, and uses our benchmark instances.

MSP due to its integrated Π minimisation, which would have not been possible with the ED_{FORM}, and would require significantly more linear constraints with the SH_{FORM}.

Lastly, we introduce a framework for Multi-Objective Resource-Aware Modulo Scheduling (**MORAMS**) in Chapter 5 to address the third aspect.

In this chapter, we present MOOVAC, an *exact* formulation that models all aspects of the problem defined in Section 2.4, allowing it to compute an optimal solution. MOOVAC is an extension of a task scheduling formulation by Venugopalan and Sinnen [90].

“MOOVAC” is an acronym for “Modulo overlap variable constraint”.

To the best of our knowledge, MOOVAC is the first ILP-based formulation to integrate the search for the optimal Π , but is also competitive in a traditional, one-candidate- Π -at-a-time setting. MOOVAC delivers high-quality results faster than both a state-of-the-art heuristic scheduler and a prior, highly-tuned ILP formulation.

We investigate strategies to improve the practicability of exact modulo scheduling, namely employing different time limits and upper bounds for the schedule length. In this context, we propose improvements to an existing bound that lead to significantly better schedule length estimates. The effects of the different time limits and bounds on the schedulers’ runtime and solution quality are evaluated in an extensive experimental study covering 188 loops from two HLS benchmark suites.

Please note that the formulations presented in this chapter assume that all shared operator types are fully pipelined internally, i.e. $\forall q \in Q^{\text{Sh}} : b_q = 1$. An extension to arbitrary, and even variable blocking times will be discussed in Section 6.3.

This chapter is based on:

- [65] Julian Oppermann, Melanie Reuter-Oppermann, Lukas Sommer, Andreas Koch and Oliver Sinnen. ‘Exact and Practical Modulo Scheduling for High-Level Synthesis’. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 12.2 (2019)

A preliminary version appeared as:

- [63] Julian Oppermann, Andreas Koch, Melanie Reuter-Oppermann and Oliver Sinnen. ‘ILP-based modulo scheduling for high-level synthesis’. In: *International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES, Pittsburgh, Pennsylvania, USA*. 2016

3.1 THE MOOVAC FORMULATION

We propose to tackle the MSP by including as much information as possible into an ILP. To this end, we integrate the Π minimisation into the formulation and obtain a bi-criteria problem that we call MOOVAC-I. As discussed in Section 2.7, there are various different approaches how to formulate a scheduling problem as an ILP, using different decision

variables. Recent work on a related scheduling problem has shown that using overlap variables is more efficient than other approaches [90], so we take this approach here. Together with a linearisation of otherwise quadratic constraints, we expect to achieve a problem structure that can be solved in reasonable time for practically relevant problem instances. In addition, the *entire* runtime is used to determine a feasible and potentially optimal solution, while a practical downside of single-II approaches is that the time spent on *unsuccessful* candidate IIs is ultimately lost.

We also introduce a non-integrated variant of the MOOVAC formulation that follows the traditional approach to modulo scheduling and models a *single* scheduling attempt for a particular candidate II. Thus, we solve a formulation with only one objective, i.e. to minimise the schedule length. Solving the MSP then requires an external driver that traverses the II search space until a feasible solution is found. This single-II variant of the MOOVAC formulation is called MOOVAC-S.

We start by defining the simpler formulation MOOVAC-S and then present the necessary extensions for obtaining the more general MOOVAC-I formulation.

3.1.1 MOOVAC-S

The problem signature of the MOOVAC-S formulation differs slightly from the general signature in Table 2.1, as the II minimisation is handled outside of the linear program that is introduced in the following sections. The bounds to the II search space, Π^{\min} and Π^{\max} , are therefore passed to an external driver, which is also responsible to return the solution's interval, Π^S . During the search, the driver chooses one or more candidate intervals II that are passed to the ILP. Inside of the ILP, II is a constant.

DECISION VARIABLES We model the problem with the decision variables shown in Table 3.1: As stated in the problem signature, the output we are seeking is a solution S that provides a start time t_i^S for every operation $i \in O$. The variables t_i directly model them in our ILP formulation. With the externally specified latency l_i , an operation i 's result will be available in time step $t_i + l_i$. The variable T captures the latest finish time across all operations.

For every shared operation $i \in O^q$ using operator type $q \in O^{\text{Sh}}$, we model the binding of one of the a_q available instances to i with an integer variable w_i , which contains an index in the range $[0, a_q - 1]$.

Due to the nature of the modulo schedule, the start time of every operation t_i can be decomposed into a multiple y_i of II plus a remainder m_i which is less than II. The start time t_i is then $t_i = y_i \cdot \Pi + m_i$.

Traditionally,
increasing candidate
intervals
 $\Pi = \Pi^{\min}, \Pi^{\min} + 1, \dots, \Pi^{\max}$ are tried.
We discuss in
Section 3.3.4 why
this is still the most
viable approach for
our set of test
instances.

Table 3.1: MOOVAC-S: Decision variables

Variable and domain		Description
T	$\in \mathbb{N}_0$	latest finish time (schedule length)
t_i	$\in \mathbb{N}_0 \quad \forall i \in O$	start time
w_i	$\in \mathbb{N}_0 \quad \forall i \in O^{\text{Sh}}$	index of operator used by operation
m_i	$\in \mathbb{N}_0 \quad \forall i \in O^{\text{Sh}}$	index of congruence class (modulo Π)
y_i	$\in \mathbb{N}_0 \quad \forall i \in O^{\text{Sh}}$	helper in congruence class computation
ω_{ij}	$\in \{0, 1\} \quad \forall i, j \in O^q, \begin{cases} i \neq j, \\ \forall q \in Q^{\text{Sh}} \end{cases}$	$\begin{cases} 1 & w_i < w_j \\ 0 & \text{otherwise} \end{cases}$
μ_{ij}	$\in \{0, 1\} \quad \forall i, j \in O^q, \begin{cases} i \neq j, \\ \forall q \in Q^{\text{Sh}} \end{cases}$	$\begin{cases} 1 & m_i < m_j \\ 0 & \text{otherwise} \end{cases}$

We define such variables for all shared operations $i \in O^{\text{Sh}}$. m_i is the congruence class (modulo Π) implied by the current start time t_i and is represented by an integer in the range $[0, \Pi - 1]$. y_i 's value is bound to the integer division $\lfloor t_i / \Pi \rfloor$.

OPERATOR LIMIT MECHANISM A valid modulo schedule must not oversubscribe any of the operator types $q \in Q^{\text{Sh}}$ in any congruence class $x \in [0, \Pi - 1]$ (cf. (2.2) in Section 2.4). A common abstraction of this condition is the **MRT** [74]. As illustrated in Figure 3.1, an MRT contains a row for each allocated operator, and a column for each congruence class (modulo the current candidate Π). Heuristic modulo schedulers often use an MRT as an explicit data structure, and successively assign operations to the cells of the table. In order for a schedule to be valid, it must be ensured that each cell is occupied by at most one operation.

While we do not use an explicit MRT in the Moovac formulation, it is still a useful intuition, as the w_i and m_i variables belonging to an operation $i \in O^q$ induce an MRT-like structure for operator type q . Our goal therefore is to model that no pair of operations $i, j \in O^q$ shares an MRT cell. Formally, we wish to enforce

$$w_i \neq w_j \vee m_i \neq m_j \quad \forall i, j \in O^q : i \neq j.$$

However, the inequality relations as well as the disjunction need to be linearised for the ILP formulation.

To this end, we introduce the following *overlap* variables on all pairs of operations $i, j \in O^q, i \neq j$ that use the same shared operator type $q \in Q^{\text{Sh}}$. The binary variable $\omega_{ij} = 1$ indicates that i 's operator index is strictly less than j 's operator index. Analogously, $\mu_{ij} = 1$ models

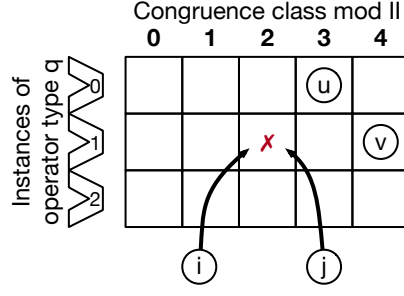


Figure 3.1: MRT as induced by the operations' w - and m -decision variables. Each operator type instance can only be used by one operation per congruence class, thus each cell can be occupied by at most one operation.

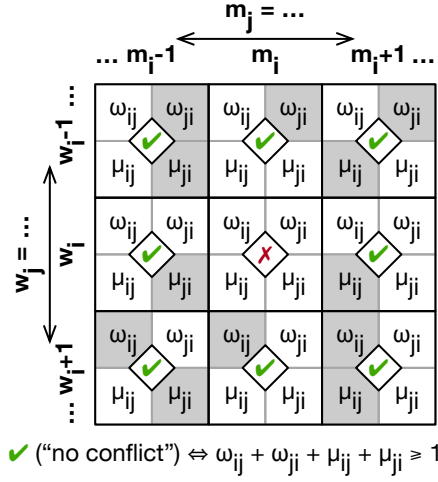


Figure 3.2: Consider two operations i and j that compete for the same MRT cell as indicated in Figure 3.1. This sketch shows the values of the overlap variables (white = '0', grey = '1') for different assignments of w_j and m_j in relation to w_i and m_i . For example, in the top right corner, we assume $w_j = w_i - 1$ and $m_j = m_i + 1$, which is an assignment that does not result in a conflict.

that i 's congruence class index is strictly less than j 's congruence class index. With these variables, we can express $w_i \neq w_j \Leftrightarrow w_i < w_j \vee w_i > w_j \Leftrightarrow \omega_{ij} + \omega_{ji} \geq 1$, and $m_i \neq m_j \Leftrightarrow \mu_{ij} + \mu_{ji} \geq 1$. It follows that i and j are not in conflict if and only if $\omega_{ij} + \omega_{ji} + \mu_{ij} + \mu_{ji} \geq 1$ is satisfied. Figure 3.2 demonstrates the interaction between two operations' w and m variables and their corresponding overlap variables.

Note that the externally specified operator allocation a_q is modelled indirectly through the range of operator indices $[0, a_q - 1]$ that can be assigned to the w_i variables, or in terms of the MRT intuition, through the number of rows that can be occupied by operations in each congruence class. In consequence, we do not explicitly count the number of operations in a particular congruence class to ensure (2.2).

CONSTRAINTS Using the decision variables defined in Table 3.1, the ILP formulation of MOOVAC-S is described by the objective (3.1) and constraints (3.2)–(3.16). The constraints related to the operator limit mechanism are defined for all pairs of different operations using the same shared operator type. For brevity, we introduce the symbol \mathcal{D} to factor out the domain specification for these constraints.

$$\text{Let } \mathcal{D} = \left\{ (q, i, j) \mid \forall q \in Q^{\text{Sh}} : \forall i, j \in O^q, i \neq j \right\}$$

$$\min \quad T \quad (3.1)$$

$$\text{s.t.} \quad t_i + l_{ij} \leq t_j + d_{ij} \cdot \Pi \quad \forall (i \rightarrow j) \in E \quad (3.2)$$

$$\omega_{ij} + \omega_{ji} \leq 1 \quad \forall (q, i, j) \in \mathcal{D} \quad (3.3)$$

$$w_j - w_i - 1 - (\omega_{ij} - 1) \cdot a_q \geq 0 \quad \forall (q, i, j) \in \mathcal{D} \quad (3.4)$$

$$w_j - w_i - \omega_{ij} \cdot a_q \leq 0 \quad \forall (q, i, j) \in \mathcal{D} \quad (3.5)$$

$$\mu_{ij} + \mu_{ji} \leq 1 \quad \forall (q, i, j) \in \mathcal{D} \quad (3.6)$$

$$m_j - m_i - 1 - (\mu_{ij} - 1) \cdot \Pi \geq 0 \quad \forall (q, i, j) \in \mathcal{D} \quad (3.7)$$

$$m_j - m_i - \mu_{ij} \cdot \Pi \leq 0 \quad \forall (q, i, j) \in \mathcal{D} \quad (3.8)$$

$$\omega_{ij} + \omega_{ji} + \mu_{ij} + \mu_{ji} \geq 1 \quad \forall (q, i, j) \in \mathcal{D} \quad (3.9)$$

$$t_i = y_i \cdot \Pi + m_i \quad \forall i \in O^{\text{Sh}} \quad (3.10)$$

$$w_i \leq a_{\sigma(i)} - 1 \quad \forall i \in O^{\text{Sh}} \quad (3.11)$$

$$m_i \leq \Pi - 1 \quad \forall i \in O^{\text{Sh}} \quad (3.12)$$

$$t_i + l_i \leq T \quad \forall i \in O \quad (3.13)$$

$$t_i \in \mathbb{N}_0 \quad \forall i \in O \quad (3.14)$$

$$w_i, y_i, m_i \in \mathbb{N}_0 \quad \forall i \in O^{\text{Sh}} \quad (3.15)$$

$$\omega_{ij}, \mu_{ij} \in \{0, 1\} \quad \forall (q, i, j) \in \mathcal{D} \quad (3.16)$$

The default objective is to minimise the schedule length (3.1).

We model the dependence edges with (3.2), which are directly adopted from the precedence constraint (2.1) in the definition of the MSP.

As both sets of overlap variables are defined by a *strictly less* relation, for a given pair of operations i, j , ω_{ij} and ω_{ji} , as well as μ_{ij} and μ_{ji} cannot be 1 at same time. This is ensured by (3.3) and (3.6).

The overlap variables are bound to their desired values by the pairs of constraints (3.4)–(3.5) and (3.7)–(3.8), respectively. For brevity, we explain their function in the context of μ_{ij} , as the constraints for ω_{ij} work analogously. (3.7) are fulfilled if $m_i < m_j$ or $\mu_{ij} = 0$, and (3.8)

are fulfilled if $m_i \geq m_j$ or $\mu_{ij} = 1$. In these constraints, the second expression uses the candidate Π as a big-M constant, meaning that its value is big enough to fulfil the constraint regardless of the rest of the expression. Due to the apparent contradictions, these constraints can only be fulfilled if and only if $m_i < m_j$ and $\mu_{ij} = 1$, or $m_i \geq m_j$ and $\mu_{ij} = 0$, resulting in the desired behaviour.

As derived above, constraints (3.9) ensure a conflict-free operator usage for every pair of operations i, j , i.e. the operations are either assigned to different operators, mapped to different congruence classes, or both.

(3.10) define the congruence class index m_i as expressed by the operation's start time t_i modulo Π .

(3.11) bound the operator instance indices for each operation using shared type q to be less than the externally specified limit a_q . Analogously, (3.12) ensure that an operation's congruence class index is less or equal to the candidate Π .

The latest finish time T is defined by (3.13) to be greater or equal to the finish times of any operation. In our implementation, we restrict this constraints to sink operations, i.e. operations that do not have outgoing forward edges ($i \rightarrow j$) with $d_{ij} = 0$.

(3.14)–(3.16) are domain constraints to enforce non-negativity respectively Boolean values for the decision variables.

3.1.2 MOOVAC-I

The MOOVAC-I formulation conforms to the problem signature in Table 2.1.

Note that the structure of the linear program as defined by the MOOVAC-S formulation is already independent of the concrete value of the candidate Π : The set of decision variables and constraints is the same, and only the numerical values in the constraints differ between scheduling attempts. This is not the case in time-indexed formulations, such as EDFORM, in which the numbers of binary variables m_i^x and constraints (cf. (2.15) in Section 2.8.1) vary with the candidate Π .

In order to integrate the Π minimisation into the MOOVAC-S formulation, we replace the formerly constant candidate Π with a new integer decision variable Π that is bounded to the Π search space as $\Pi^{\min} \leq \Pi \leq \Pi^{\max}$. However, integrating the Π minimisation naively has a major drawback: It results in quadratic (i.e. containing a multiplication of decision variables) constraints (3.7), (3.8) and (3.10).

We linearise constraints (3.7), (3.8) by replacing the occurrence of Π with the upper bound of the Π search space, Π^{\max} :

$$m_j - m_i - 1 - (\mu_{ij} - 1) \cdot \Pi^{\max} \geq 0 \quad \forall (q, i, j) \in \mathcal{D} \quad (3.17)$$

$$m_j - m_i - \mu_{ij} \cdot \Pi^{\max} \leq 0 \quad \forall (q, i, j) \in \mathcal{D} \quad (3.18)$$

This has no effect on the constraints' functionality, as the interval value is used as a big-M constant here.

The remaining quadratic constraints (3.10) are broken down into individual constraints for all possible values of y_i . To this end, we need to introduce an upper bound y^{\max} for the y_i variables. Recall that the y_i variables represent the value $\lfloor t_i/\Pi \rfloor$ in the calculation of operation i 's congruence class. Given an upper bound T^{\max} for the schedule length (and in consequence, for all t_i), and using the Π search space's lower bound Π^{\min} , the y_i variables are bounded by $y^{\max} = \lfloor T^{\max}/\Pi^{\min} \rfloor$.

With this bound and introducing new binary variables $\psi_i^{(x)}$ for $0 \leq x \leq y^{\max}$ and all operations using a shared operator type, constraints (3.10) are replaced by:

$$\psi_i^{(x)} = 1 \rightarrow t_i = x \cdot \Pi + m_i \quad \forall x \in [0, y^{\max}] : \forall i \in O^{\text{Sh}} \quad (3.19)$$

$$\sum_{x=0}^{y^{\max}} \psi_i^{(x)} \geq 1 \quad \forall i \in O^{\text{Sh}} \quad (3.20)$$

$$\sum_{x=0}^{y^{\max}} x \cdot \psi_i^{(x)} = y_i \quad \forall i \in O^{\text{Sh}} \quad (3.21)$$

$$\psi_i^{(x)} \in \{0, 1\} \quad \forall i \in O^{\text{Sh}}, \forall x \in [0, y^{\max}] \quad (3.22)$$

The indicator constraints (3.19) conditionally model the modulo decomposition for every possible value of y_i . Constraints (3.20) force that at least one of these linearised decompositions is selected for the solution. Note that the \geq -constraints are sufficient here, as at most one $\psi_i^{(x)}$ can be non-zero due to the mutually-exclusive nature of the decomposition. Constraints (3.21) define the value of y_i according to the selected decomposition. These constraints are not required for the correctness of the MOOVAC-I formulation, but make solutions interchangeable between MOOVAC-I and MOOVAC-S, a property we leverage in Section 3.2.1. Lastly, constraints (3.22) are the domain constraints for the binary variables representing the possible values of y_i .

Note that a similar extension would be possible for the SHFORM, but would incur a higher complexity: as the modulo decomposition is interwoven with the modelling of the dependence edges (cf. (2.24) in Section 2.8.2), the linearisation would concern all edges, instead of the operations using shared operator types. Table 3.3 provides a statistical breakdown of the instances used in our evaluation, and suggests that $|E| \ll |O^{\text{Sh}}|$.

The MSP's bi-criteria objective can now be modelled directly in the MOOVAC-I formulation, which is defined by:

$$\min \Pi \quad (3.23)$$

$$\min T \quad (3.24)$$

$$\text{s.t. } (3.2) - (3.6), (3.9), (3.11) - (3.16), (3.17) - (3.22)$$

We discuss possible upper bounds in Section 3.2.2.

An indicator constraint is of the form $\langle \text{bin} \rangle = \langle \text{val} \rangle \rightarrow \langle \text{cons} \rangle$: if the binary variable $\langle \text{bin} \rangle$ has the value $\langle \text{val} \rangle$, the right-hand side constraint $\langle \text{cons} \rangle$ must be satisfied; otherwise, it may be violated [36]. Indicator constraints are supported natively in modern ILP solvers.

3.2 STRATEGIES FOR MODULO SCHEDULING IN PRACTICE

We now discuss the use of time limits, and bounds on the schedule length. A time limit allows to cap the worst-case solution times, whereas a schedule length bound is required to make the integration of the II search in the Moovac-I formulation possible, and can help to narrow down the solution space.

3.2.1 Time-limited Scheduling

Since modulo scheduling is an NP-hard problem [52], we cannot rule out the possibility to encounter problem instances that lead to exponential runtimes when solved with the exact approaches, or cause the heuristic approach to get stuck.

To this end, we impose a time limit λ per candidate II. For all approaches, this includes the time κ to construct the linear program via the solver's API. Combining the exact approaches with a time limit can lead to non-optimal solutions, however, we retain the ability to qualify how close to optimality the returned solutions are, as discussed in the following paragraphs.

OPTIMAL SOLUTION(S) Our optimisation objective for a given MSP instance, as defined in Section 2.4, is to find a solution S which lexicographically minimises the tuple (II^S, T^S) . Note that multiple optimal solutions may exist according to this objective. For example, consider a shared operator type that only provides one instance. The operations using this type need to be sequentialised, however, the actual order might make no difference to the resulting II and schedule length.

We call a solution S *optimal* if it achieves the minimal interval II^* , and represents a schedule that has the minimal length T^* among all solutions with II^* .

MOOVAC-S, EDFORM Each scheduling attempt with the single-II Moovac formulation as well as with the EDFORM is executed as one invocation of the ILP solver, so a time limit of $\lambda - \kappa$ can directly be specified via its API.

When the solver returns from a scheduling attempt for a candidate interval, II, it reports one of the following outcomes:

- **Optimal.** A schedule was found, and the solver *proved* that its length is optimal.
- **Feasible.** A schedule was found, but, within the time limit, the solver was unable to determine whether its length is optimal.
- **Unknown.** No schedule was found within the time limit. We conservatively consider the candidate II to be infeasible.

- Infeasible. The solver proved that the candidate Π is infeasible.

Now let Π^S be the smallest candidate Π for which at least a feasible modulo schedule was found. Π^S is the optimal Π if, trivially, $\Pi^S = \Pi^{\min}$, or if all candidates $\chi \in [\Pi^{\min}, \Pi^S)$ are known to be infeasible. In case scheduling for Π^S yielded an optimal schedule length, we have also found an overall optimal solution, otherwise we only have $\Pi^S = \Pi^*$.

MODULOSDC Recall from Section 2.7.3 that a scheduling attempt with the MODULOSDC algorithm necessitates constructing and solving multiple linear programs. We thus do not impose our time limit on the individual invocations of the LP solver, but instead on the scheduling attempt as a whole, i.e. including the solver runtimes and all MRT and backtracking operations.

In case the current candidate Π is infeasible, the algorithm would potentially run until all possible schedules were inspected. To that end, the algorithm maintains a budget of $6 \cdot |O|$ backtracking steps [7] to provide another failsafe for fruitless scheduling attempts. This mechanism is still in place in our implementation, as otherwise, even simple problem instances would deplete the whole time budget on infeasible candidate Π s.

The only situation in which an interval Π^S returned by the MODULOSDC scheduler is known to be optimal is if $\Pi^S = \Pi^{\min}$, as the algorithm can only run out of time or backtracking steps, but never prove infeasibility of a candidate Π .

The algorithm contains no means to determine the optimality of the found schedule length.

MOOVAC-I The integration of the Π search in the MOOVAC-I formulation enables a different solution strategy. Instead of performing the minimisation of the first objective (= the initiation interval) by iteratively solving and minimising multiple ILPs for the second objective (= here, the schedule length), we can optimise both objectives directly using a single MOOVAC-I-ILP.

ILP solvers handle multiple objectives either by attempting to solve the problem only once with user-specified weights for the objective functions, or by addressing the different objectives in individual steps, according to priorities given by the user. The latter, lexicographic approach is more suitable in the context of the modulo scheduling problem, because practitioners would almost certainly choose weights that strongly favour better Π s anyway. Some ILP solvers offer an API to perform multi-objective optimisation automatically. However, in order for our approach to be solver-independent, and because it makes it easier to reason about the solution quality in the presence of time limits, we implement the lexicographic approach ourselves.

To this end, we begin by setting a time limit of λ and instruct the solver to only minimise the Π . When the solver returns, it reports one of the following outcomes:

- Optimal. A solution S was found, and its interval Π^S was *proven* to be optimal, i.e. $\Pi^S = \Pi^*$.
- Feasible. A solution S was found, but the solver was unable to determine whether its interval Π^S is optimal within the time limit. We only know that Π^S is optimal if it is equal to Π^{\min} .
- Unknown. No solution was found within the time limit; give up and report failure.

The interval computed in the first step, Π^S , is fixed during the second step. This allows us to reduce the complexity of the current ILP by “downgrading” it to resemble the MoovAC-S formulation. We add a constraint to bind the decision variable Π to Π^S , and replace the constraints (3.19)-(3.22) by the simpler MoovAC-S version of the modulo decomposition, $t_i = y_i \cdot \Pi^S + m_i, \forall i \in O^{\text{sh}}$.

Again with a time limit of λ , the solver is now instructed to minimise only the schedule length. Note that this is a *warm start* made possible by constraints (3.21), meaning the feasible solution from the first step is still valid for the modified ILP, and the solver will work on improving it. Therefore, only two outcomes are possible after the solver returns the second time:

- Optimal. The solver *proved* that the current schedule’s length is optimal.
- Feasible. Within the time limit, the solver was unable to prove whether the current schedule length is optimal.

The reasoning about the optimality of the solution S is straightforward: We have an optimal solution only if both steps yielded an optimal result according to their respective objective. If optimality can be proven in the first step, but not in the second step, we trivially know $\Pi^S = \Pi^*$. It is possible that we just find a feasible interval, but determine the optimal schedule length for that interval. This situation is counted as an overall feasible solution.

FALLBACK SCHEDULE Recall that we save a resource-constrained non-modulo schedule from the computation of Π^{\max} . This schedule serves as a fallback in the unfortunate case that all modulo scheduling attempts fail, and allows the compilation to continue after a guaranteed maximum time.

3.2.2 Bounded Schedule Length

Imposing an upper bound T^{\max} on the schedule length is beneficial for practical modulo scheduling, as it reduces the overall solution space of

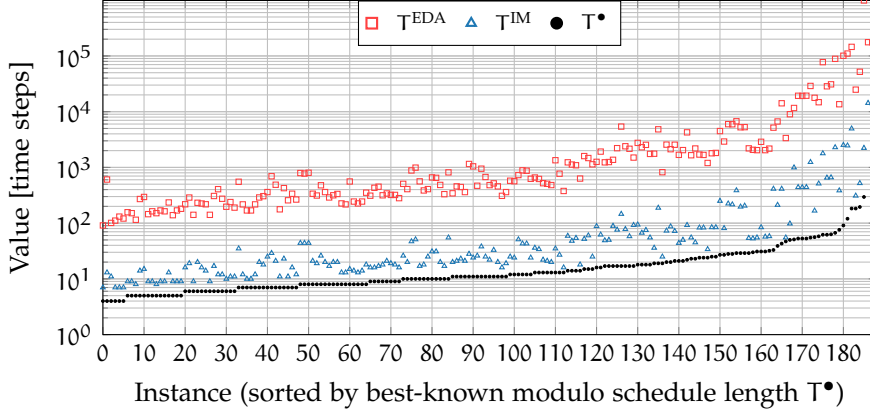


Figure 3.3: Distribution of modulo schedule lengths, and their bounds. Note the logarithmic scale on the Y-axis.

the combinatorial problem, and helps the LP solver to give up earlier on fruitless branches in situations where the infeasibility of a partial solution is not obvious to the solver, causing it to try increasingly late start times for operations in the search. Additionally, such a bound is required to make the proposed linearisation of constraints (3.19) in the MOOVAC-I formulation possible.

While we desire T^{\max} to be as tight as possible, any chosen bound needs to be an overestimate, i.e. higher or equal to the optimal schedule length for a given Π , in order to prevent the solver to incorrectly rule out that Π as infeasible.

CONSERVATIVE BOUND Eichenberger, Davidson and Abraham [27] proposed and proved the following upper bound for the length of a modulo schedule, which we adapt here to our notation. Let $\Delta = \max_{(i \rightarrow j) \in E} l_{ij}$, i.e. the maximum number of time steps that two operations connected by a dependence edge must be started apart. We define

$$T^{\text{EDA}} = |O| \cdot (\Delta + \Pi^{\max} - 1). \quad (3.25)$$

Intuitively, each operation $i \in O$ requires at most Δ time steps before the next dependent operation can start, but may need to be deferred for up to $\Pi - 1$ time steps due to the modulo operator constraints. We approximate the actual candidate Π with its upper bound Π^{\max} here, because this slightly looser bound retains the same numerical value across all scheduling attempts.

IMPROVED BOUND Figure 3.3 shows the distribution of best-known schedule lengths, and corresponding bound values for T^{EDA} , for the test instances used in our experimental evaluation (Section 3.3.4). It is obvious that T^{EDA} overestimates the actual schedule length by roughly an order of magnitude. We propose two improvements to

Eichenberger's bound that result in a safe upper bound T^{IM} that is much closer to the actual modulo schedule length (also shown in Figure 3.3).

Let $\Delta_i = \max_{j \in O: (i \rightarrow j) \in E} l_{ij}$ denote the maximum number of time steps that any successor j of i needs to be started after i . We define the improved bound as

$$T^{\text{IM}} = \underbrace{\sum_{i \in O} \Delta_i}_{\text{a) assume sequential schedule}} + \underbrace{\sum_{q \in Q^{\text{Sh}}} \sum_{x=0}^{|O^q|-1} \left\lfloor \frac{x}{a_q} \right\rfloor \cdot b_q}_{\text{b) account for shifts due to modulo operator conflicts}}. \quad (3.26)$$

This definition keeps the basic ideas in Eichenberger's bound of a) assuming that in the worst case, all operations need to be scheduled sequentially, and b) of accounting for the need to shift operations to a later start time due to the modulo operator conflicts.

Our improvement to a) is straight-forward: Instead of assuming $|O|$ -many same-sized windows of time steps that are large enough to accommodate every operation in the MSP, we determine the maximum number of time steps individually for each operation.

We now quantify the worst-case number of modulo operator conflicts more precisely to derive a tighter estimate for b). First, unlimited operations never need to be shifted as, by definition, they never compete for operators. Next, recall the intuition of the MRT (Figure 3.1) for an operator type q with one available instance and a blocking time of $b_q = 1$, and imagine it is successively filled with the operations $\{i_0, \dots, i_{|O^q|-1}\} = O^q$ competing for that q -instance during scheduling. Operation i_0 will not need to be shifted as it is the first operation to be assigned to the MRT. i_1 may need to be shifted for at most one time step if it conflicts with i_0 . i_2 may need to be shifted for at most two time steps if it conflicts with i_0 , and then with i_1 , which was already shifted by one time step to resolve its own conflict with i_0 . In the worst case, all operations in O^q are initially in conflict for using the q -instance in the same congruence class. In general, an operation i_x will need to be shifted for at most x time steps, i.e. equal to the number of operations that are already assigned to the MRT.

This reasoning is easily extended to operator types q that provide $a_q > 1$ available instances, and have blocking times $b_q > 1$. In that case, the required conflict-resolving shift amount is equal to b_q , and increases to the next multiple of b_q only every a_q operations, which means that an operation i_x will need to be shifted at most $\left\lfloor \frac{x}{a_q} \right\rfloor \cdot b_q$ time steps in the worst case.

Summing the maximum shift amounts for all operations over all shared operator types, we arrive at the expression in Eq. (3.26-b).

Note that we neither make any assumptions about the particular conflict-resolving shift amount an individual operation might require,

Note that while the ILP-based schedulers discussed in this work do not operate in such a way internally, it is still a helpful conceptual model.

nor about the order in which operations are assigned to the MRT, but rather estimate the maximum number of shifts to be expected for a set O^q as a whole.

3.3 EXPERIMENTAL EVALUATION

We now compare scheduler implementations based on MOOVAC-S, MOOVAC-I and EDFORM, and the MODULOSDC scheduler, on a large set of typical high-level synthesis loops, with regards to the *scheduler runtime* τ and the *quality of results* in terms of the interval II^S and the schedule length T^S of the computed solution S .

3.3.1 Compiler Context

We implemented all schedulers in the Nymble HLS compiler [42]. Nymble is based on the LLVM framework [53], version 3.3 and uses the framework’s analyses and optimisations.

All function calls in the benchmark programs are inlined exhaustively. The resulting modules are optimised with LLVM’s preset -O2, but without performing loop unrolling.

INTERMEDIATE REPRESENTATION Nymble uses a hierarchical CDFG as its main IR, i.e. the compiler constructs a CDFG for each natural loop in the input program. Within such a graph, control information is translated to predicated data-flow, and nested loops are modelled as operations using a special operator type. This IR allows us to attempt to modulo schedule loops on all nesting levels and with arbitrary control structures, without the need to perform preparative transformations such as if-conversion.

We rely on LLVM’s dependence analysis to discover intra- and inter-iteration memory dependences. These dependences are encoded as additional edges in the CDFG, and handled uniformly by the schedulers. Due to a technical limitation, we currently consider all of these backedges to express loop-carried dependences on the *immediately preceding* iteration, leading to conservatively larger IIs. However, we expect the impact on our results to be small, as in our compiler context, the dependence analysis could only determine exact backedge distances $d_{ij} > 1$ for less than 0.3 % of the backedges that were constructed.

The schedulers used the operator allocation in Table 3.2.

3.3.2 Reference Schedulers

We implemented the MODULOSDC scheduler according to [7], but used a simpler height-based priority function, and changed the objective to

Table 3.2: Allocation for shared operator types

Operator type	# available
Memory Load/Store	1 each
Nested loops	1
Integer Div/other	8/ ∞
FP Add/Sub/Mul/other	4/4/4/2 each

Table 3.3: Problem sizes

Metric	min.	med.	avg.	max.
# operations	19	54	125	2654
# shared ops.	0	4	12	221
# forward edges	28	81	245	6752
# backedges	2	4	42	1222

a schedule length minimisation. Our implementation of the EDfORM mirrors Section 2.8.1.

The data-flow graphs constructed by Nymble contain a unique start operation that is required to be scheduled to time step 0. A corresponding constraint is added to all schedulers.

3.3.3 Test Setup

We used Gurobi 8.0 as (I)LP solver for all schedulers.

The experiments were conducted on 2x12-core Intel Xeon E5-2680 v3 systems running at 2.8 GHz with 64 GiB RAM. The schedulers were allowed to use up to 8 threads and 16 GiB of memory per loop.

Each experiment was repeated three times to compensate for varying system load, and for each loop, we include in our evaluation the result of the “best” modulo scheduling attempt with regard to the smallest II, schedule length, and lastly, scheduling runtime. The scheduling runtime always includes the time to construct the respective linear programs via the Gurobi API. The generated schedules were verified by RTL simulation of the hardware accelerators generated by Nymble.

3.3.4 Test Instances

The modulo scheduling test instances, i.e. loops/graphs, used in this evaluation originate from the HLS benchmark applications in the CHStone [38] and MachSuite [75] collections. We excluded backprop, bfs/bulk, fft/transpose and nw from MachSuite due to limitations

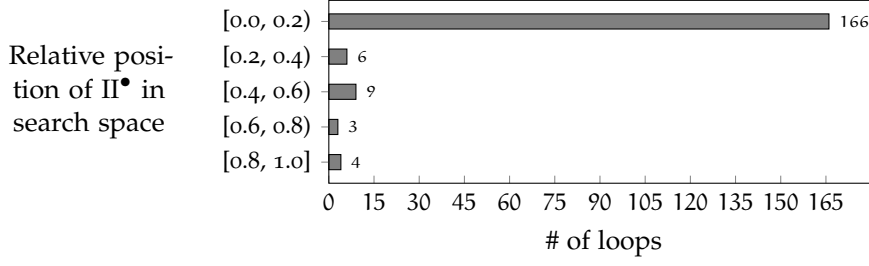


Figure 3.4: Histogram of the number of loops according to the relative position of their Π^* within the Π search space, i.e. $0 = \Pi^{\min}$ and $1 = \Pi^{\max}$.

in Nymble that prohibited the synthesis of these applications even without using the modulo schedulers evaluated here. Also, we removed `printf` statements inside the computational kernels of CHStone’s `aes` and `jpeg` applications.

In total, we obtained 354 loops to modulo schedule. Due to the exhaustive inlining performed in Nymble, and the presence of small, idiomatic loops (e.g. array initialisations), we detected that 166 loops are identical or isomorphic to other loops.

The remaining 188 loops comprise our set of test instances used in the rest of this evaluation. Table 3.3 characterises the sizes of the corresponding MSPs. The histogram in Figure 3.4 shows the distribution of the best-known intervals Π^* for the test instances, normalised to the range between 0 ($= \Pi^{\min}$) and 1 ($= \Pi^{\max}$). In summary, the majority of loops in our test set have a best-known Π equal or close to the lower bound of the search space. To this end, the single- Π approaches in this evaluation traverse the Π search space in the traditional ascending order. In order to restrict the total runtime per instance in our experiment to 24 hours, we limit the search to at most 23 candidate Π s for the single- Π approaches, i.e. we attempt to schedule for $\Pi^{\min}, \Pi^{\min} + 1, \Pi^{\min} + 2, \dots, \Pi^{\min} + 22$. All loops in our evaluation have $\min(\Pi^{\max}, \Pi^*) \leq \Pi^{\min} + 22$, i.e. this attempt limit did not preclude any of the schedulers from finding a feasible solution.

3.3.5 Comparison of Approaches, Time Limits and Bounds

We schedule the test instances with the MOOVAC-S formulation (denoted by MvS), the MOOVAC-I formulation (MvI), the ED_{FORM} (ED_F), and the MODULO_{SDC} algorithm (MSDC).

We investigate time limits (Section 3.2.1) τ of 1, 5, 15 and 60 minutes, and impose either no upper bound on the schedule length (denoted by ∞), or use the bounds T^{EDA} or T^{IM} from Section 3.2.2.

While the time limit and schedule length bound seem independent on first sight, they both limit the computational effort that the solver

spends on a particular scheduling attempt. In the case of the schedule length bound, this limit works indirectly by reducing the branch-and-bound solution space.

To this end, we present the results of modulo scheduling with the different time limits and upper bounds together, both in terms of scheduler runtimes (Tables 3.4 to 3.5) and schedule quality (Tables 3.6 to 3.7).

METHODOLOGY Our methodology here is twofold: On the one hand, we provide accumulated runtimes and quality measures for the computed initiation intervals across all instances as intuitive metrics for the performance of a certain configuration $K = \text{SCH}_{T^{\max}}^{\lambda}$, i.e. a combination of a scheduling approach “SCH”, a bound T^{\max} for the schedule length, and a time limit λ . However, this presentation disregards that all instances are in fact *independent* of each other. To this end, we additionally count the number of instances for which a configuration achieves a particular result, e.g. to schedule an instance between 10 and 100 seconds, or to prove optimality for a solution. This allows us to assess to which degree a given configuration is practical on our representative benchmark set, and highlights the presence of outlier instances (i.e. with exceptionally long runtimes or low quality schedules) as well as their influence to the accumulated metrics. Note that outliers have to be expected for every exact modulo scheduling approach due to the NP-hard nature of the underlying problem. However, if their number is small, this does not automatically impair the scheduler’s practicability. Additionally, a benefit of the ILP method is that a quality measure (i.e. the branch-and-bound method’s “gap” value [48]) is available for any feasible solution, which allows the practitioner to decide to try another scheduler (exact or heuristic) for the same problem.

We introduce the notion of the *best-known* interval Π^{\bullet} and schedule length T^{\bullet} to indicate the best solution found across all experiments conducted in this work, i.e. including all approaches and configurations, as a reference point for loops for which the optimal solution is not known.

MOOVAC The Moovac-based schedulers outperform the other approaches in this experiment: The overall fastest configuration is $\text{MvI}_{T^{\text{IM}}}^{1\text{min}}$ with an accumulated runtime of 15.7 minutes for all 188 test instances. At first glance, the $\text{MvS}_{T^{\text{IM}}}^{\leq 15\text{min}}$ configurations, which find the highest-quality solutions overall, seemingly have a significant advantage concerning the result quality over $\text{MvI}_{T^{\text{IM}}}^{\leq 15\text{min}}$, but note that the sum of Π differences (column “ $\Pi^S - \Pi^{\bullet} : \Sigma$ ”) is concentrated on only three loops (column “ $\Pi^S - \Pi^{\bullet} : \#$ ”). For all other loops, $\text{MvI}_{T^{\text{IM}}}^{\leq 15\text{min}}$ delivers results on par with its single- Π variant (columns under “# loops with...”).

$MvS_{T^{IM}}$ and $MvI_{T^{IM}}$ achieve the best trade-off between scheduler runtime and result quality already with the 1 minute time limit. The runtimes increase with the time budget, but the result quality is only improved marginally. In the 60 minute configurations, the solution space covered by the MOOVAC-based schedulers for some instances becomes too large to fit in the 16 GiB allocated to the experiments (column “OOM”).

$MvS_{T^{IM}}$ clearly benefits, in terms of runtime and quality, from the presence of an upper bound for the schedule length, and from the tighter estimate that T^{IM} gives compared to T^{EDA} . This effect is even more pronounced for $MvI_{T^{IM}}$, because the value of the bound directly affects the complexity of the MOOVAC-I-ILP.

EDFORM Being an exact scheduler as well, the EDFORM achieves optimal results for almost as many loops as the MOOVAC-based approaches. However, especially with the smallest time budget, it is unable to find modulo schedules for several large instances (column “no S”), and runs out of memory for one loop. Given more time, the solution quality improves, but unfortunately four more loops cannot be scheduled within the 16 GiB memory limit. The accumulated runtimes and solution quality is mostly unaffected by the choice of schedule length bound.

MODULOSDC As expected, the MODULOSDC algorithm is quite fast, but lacks the capability to prove optimality of its solutions, and misses the best-known IIs by at least 77, spread over 20 instances. Starting with the 5 minute configuration, imposing an upper bound on the schedule length effectively serves as a third measure (besides time limit and backtracking budget) to give up on fruitless scheduling attempts, and limits the overall runtime to roughly 90 minutes. The quality metrics remain unchanged over the four different time limits. Curiously, without a bound, one more loop can be scheduled, which alone contributes an II difference of 77 to $MSDC_{T^{EDA}}$ and $MSDC_{T^{IM}}$. The length of the schedule found by $MSDC_{\infty}$ for this particular instance does not come near the values of the bounds, though.

CONSTRUCTION TIME The maximum ILP construction time κ per scheduling attempt is below 10 seconds for the exact approaches, and turned out to be negligible compared to the runtime of the solver’s branch-and-bound algorithm. This excludes loops for which a scheduling attempt ran out of memory, because we did not record the precise point in time when the memory limit was hit. We also did not record a separate construction time for the MODULOSDC scheduler, as the linear program is constantly changed over the course of the algorithm.

PERFORMANCE PROFILE The performance profile [21] in Figure 3.5 shows a different view on the dataset for the 5 minute experiment, as it relates the runtimes of different configurations for each individual loop: for every configuration K and loop (i.e. MSP instance) J , the ratio

$$\frac{\tau_K(J)}{\min_{(\text{config}, K')} \tau_{K'}(J)} \quad (3.27)$$

of the scheduler runtime with K and the fastest runtime across all configurations, is computed. In the plot, we count the number of loops for which this ratio is less or equal than some factor X for a configuration K , i.e. the number of loops whose runtime with K is at most X times worse than the best-known runtime for the loop. The performance profile can be interpreted as the probability that a configuration is able to modulo schedule a loop within a certain performance envelope. The higher the curve the better. For example, the data point marked with a dark blue square on the $\text{MSDC}_{\text{TEDA}}^{5\min}$ plot signifies that the scheduler runtime for 92 loops ($= 92/188 \approx 49\%$) was at most twice as long as the fastest runtime across all configurations in the performance profile. The intersection of the plots with the “ $X = 1$ ”-line shows the number of loops for which a configuration sets the fastest runtime. This data is repeated in the table below the performance profile.

Dividing the loop counts by the total number of loops yields a probability value.

We observe that $\text{MvS}_{\text{TIM}}^{5\min}$ is the fastest configuration to schedule 149 of the 188 loops ($\approx 79\%$), followed by $\text{MvI}_{\text{TIM}}^{5\min}$ with 18 fastest runtimes. In general, the plots for the MOOVAC-based configurations (excluding $\text{MvS}_{\text{TEDA}}^{5\min}$) and for the MSDC scheduler rise quickly, meaning that they are close to the fastest runtime for the majority of instances. The slope of the plots for EDF indicates that for roughly 75 % of the loops, these configurations require at least 5x longer than the respective fastest configuration.

MOOVAC-S vs. MOOVAC-I The performance profile also hints at a key difference between the MOOVAC-S and MOOVAC-I formulations. The single-II variant is often faster for an individual loop due to the simpler ILP formulation and the location of the first feasible interval in the II search space (cf. Figure 3.4). However, the practical advantage of MvI is that its runtime is capped at $2 \cdot \lambda$ by design, which is reflected in the much faster accumulated runtimes shown in Tables 3.4 to 3.5. As discussed above, the quality metrics in Tables 3.6 to 3.7 show that this novel way of approaching the modulo scheduling problem comes with almost no loss in solution quality, with MvI_{TIM} missing the best-known interval II^\bullet for at most two additional instances compared to MvS_{TIM} .

As MOOVAC-I does not iterate over candidate IIs, it spends at most one time budget λ on the II minimisation, and one λ on optimising the secondary objective.

Table 3.4: Scheduling times for combinations of approaches, time limits and bounds

Configuration K		Loops classified by scheduler runtime τ (188 loops)											
		< 10 s		10-100 s		100-1k s		1k-10k s		$\geq 10k$ s		all	
		#	Σ	#	Σ	#	Σ	#	Σ	#	Σ	Σ [min]	
<i>Time limit: 1 minute</i>													
MvS	∞	168	0.7	8	5.9	12	92.0	-	-	-	-	98.5	
	τ^{EDA}	173	0.5	5	2.7	10	82.9	-	-	-	-	86.1	
	τ^{IM}	174	0.4	5	2.8	9	71.3	-	-	-	-	74.5	
MvI	τ^{EDA}	155	0.7	32	26.1	1	2.0	-	-	-	-	28.7	
	τ^{IM}	171	0.5	15	11.2	2	4.0	-	-	-	-	15.7	
EDf	∞	165	1.2	10	6.1	8	60.0	5	112.6	-	-	179.8	
	τ^{EDA}	167	1.5	7	3.0	9	55.8	5	112.7	-	-	173.0	
	τ^{IM}	167	1.3	9	5.5	7	52.1	5	112.7	-	-	171.7	
MSDC	∞	172	0.7	7	4.6	8	46.8	1	17.1	-	-	69.2	
	τ^{EDA}	172	1.0	8	5.5	7	44.1	1	16.8	-	-	67.5	
	τ^{IM}	172	1.0	8	5.4	7	44.4	1	16.9	-	-	67.8	
<i>Time limit: 5 minutes</i>													
MvS	∞	168	0.6	3	0.9	8	55.2	9	378.3	-	-	435.0	
	τ^{EDA}	173	0.5	3	0.7	4	32.5	8	339.5	-	-	373.1	
	τ^{IM}	174	0.4	3	0.8	3	25.1	8	292.8	-	-	319.1	
MvI	τ^{EDA}	155	0.7	11	4.2	22	117.1	-	-	-	-	122.0	
	τ^{IM}	172	0.7	5	1.8	11	70.3	-	-	-	-	72.7	
EDf	∞	165	1.2	10	6.4	5	39.8	8	520.1	-	-	567.6	
	τ^{EDA}	167	1.5	8	4.4	5	45.6	8	538.0	-	-	589.4	
	τ^{IM}	167	1.3	9	5.0	4	34.8	8	552.0	-	-	593.2	
MSDC	∞	172	0.7	7	4.6	6	23.8	3	94.9	-	-	124.1	
	τ^{EDA}	172	1.1	8	5.6	6	30.5	2	49.0	-	-	86.1	
	τ^{IM}	172	1.1	8	5.6	6	30.9	2	48.1	-	-	85.5	

The loops are classified into time brackets according to the scheduler runtime τ with a configuration K. Columns “#” show the number of loops that fell into a particular time bracket. Columns “ Σ ” show the accumulated scheduling time (in *minutes*) for the loops within a time bracket. Loops that caused the scheduler to run out of memory (cf. Tables 3.6 to 3.7) are accounted for with the maximum available time to them, i.e. $(\min(\Pi^{\max}, \Pi^{\min} + 22) - \Pi^{\min} + 1) \cdot \lambda$ for the single- Π approaches, and λ for MvI.

Table 3.5: Scheduling times for combinations of approaches, time limits and bounds (continued)

Configuration K		Loops classified by scheduler runtime τ (188 loops)										
		< 10 s		10-100 s		100-1k s		1k-10k s		≥ 10 k s		all
		#	Σ	#	Σ	#	Σ	#	Σ	#	Σ	Σ [min]
Time limit: 15 minutes												
MvS	∞	168	0.7	3	1.0	5	62.4	11	952.1	1	225.0	1241.1
	T^{EDA}	173	0.5	3	0.7	3	37.5	8	797.1	1	225.0	1060.8
	T^{IM}	174	0.4	3	0.8	2	30.0	8	655.0	1	225.0	911.2
MvI	T^{EDA}	154	0.5	12	4.5	20	268.2	2	49.0	-	-	322.2
	T^{IM}	171	0.5	6	2.0	8	120.3	3	90.1	-	-	212.9
EDF	∞	165	1.2	10	6.8	3	38.7	6	388.4	4	1058.5	1493.5
	T^{EDA}	167	1.5	9	6.0	1	15.7	7	436.6	4	1064.3	1524.1
	T^{IM}	167	1.3	9	5.1	2	22.8	6	407.5	4	1074.4	1511.2
MSDC	∞	172	0.7	7	4.8	6	24.1	3	180.7	-	-	210.3
	T^{EDA}	170	0.8	10	5.9	6	31.3	2	54.1	-	-	92.1
	T^{IM}	170	0.7	10	5.8	6	31.2	2	52.8	-	-	90.6
Time limit: 60 minutes												
MvS	∞	168	0.7	3	0.9	1	2.4	6	422.8	10	5614.8	6041.7
	T^{EDA}	173	0.5	3	0.7	1	7.3	2	120.0	9	4996.2	5124.7
	T^{IM}	174	0.4	3	0.8	-	-	4	250.0	7	4380.1	4631.3
MvI	T^{EDA}	155	0.7	11	4.3	4	27.2	18	1140.2	-	-	1172.4
	T^{IM}	172	0.7	5	1.8	-	-	11	720.9	-	-	723.4
EDF	∞	165	1.2	10	6.7	1	8.1	6	407.1	6	4877.1	5300.2
	T^{EDA}	167	1.5	9	5.7	-	-	6	508.8	6	4886.9	5403.0
	T^{IM}	167	1.4	9	5.3	1	7.4	5	373.4	6	4753.0	5140.4
MSDC	∞	172	0.8	7	4.9	6	26.1	2	52.8	1	420.4	505.0
	T^{EDA}	171	0.9	9	5.8	6	32.7	2	49.6	-	-	89.0
	T^{IM}	172	1.1	8	5.8	6	32.3	2	49.4	-	-	88.6

Table 3.6: Schedule quality for combinations of approaches, time limits and bounds

Configuration K		# loops with ...								$\Pi^S - \Pi^\bullet$	
		$\Pi^*\square$	Π^*	Π^\bullet	$\Gamma^*\square$	Γ^*	Γ^\bullet	no S	OOM	Σ	#
<i>Time limit: 1 minute</i>											
MvS	∞	173	183	184	171	182	183	2	-	5	2
MvS	Γ^{EDA}	178	183	184	176	182	183	2	-	4	2
MvS	Γ^{IM}	179	184	185	177	182	182	2	-	2	1
MvI	Γ^{EDA}	167	176	177	166	176	176	11	-	328	9
MvI	Γ^{IM}	179	182	183	177	181	181	5	-	52	3
EDF	∞	173	176	176	173	176	176	9	1	120	10
EDF	Γ^{EDA}	174	176	176	174	176	176	9	1	115	10
EDF	Γ^{IM}	175	177	177	174	176	176	9	1	114	9
MSDC	∞	146	166	166	-	152	152	3	-	77	20
MSDC	Γ^{EDA}	143	163	163	-	143	143	4	-	141	23
MSDC	Γ^{IM}	143	163	163	-	143	143	4	-	141	23
<i>Time limit: 5 minutes</i>											
MvS	∞	174	183	184	172	182	183	2	-	3	2
MvS	Γ^{EDA}	179	183	184	177	183	184	2	-	3	2
MvS	Γ^{IM}	179	184	185	177	183	183	2	-	1	1
MvI	Γ^{EDA}	170	177	178	168	177	177	9	-	246	8
MvI	Γ^{IM}	179	182	183	177	182	183	4	-	16	3
EDF	∞	176	181	181	176	180	180	1	5	34	5
EDF	Γ^{EDA}	176	181	181	176	181	181	1	5	36	5
EDF	Γ^{IM}	177	181	181	177	181	181	1	5	40	5
MSDC	∞	146	166	166	-	152	152	3	-	77	20
MSDC	Γ^{EDA}	143	163	163	-	143	143	4	-	141	23
MSDC	Γ^{IM}	143	163	163	-	143	143	4	-	141	23

The 188 loops are counted according to properties of the solutions (see Table 3.8) computed by a given approach.

Table 3.7: Schedule quality for combinations of approaches, time limits and bounds (continued)

Configuration K		# loops with ...							$\Pi^S - \Pi^\bullet$		
		$\Pi^*\square$	Π^*	Π^\bullet	$\Gamma^*\square$	Γ^*	Γ^\bullet	no S	OOM	Σ	#
Time limit: 15 minutes											
MvS	∞	174	183	184	172	183	184	2	-	2	2
	Γ^{EDA}	179	183	184	177	183	184	2	-	3	2
	Γ^{IM}	179	184	185	177	183	183	2	-	1	1
MvI	Γ^{EDA}	172	179	180	170	179	180	6	-	239	6
	Γ^{IM}	179	182	183	177	182	183	4	-	12	3
EDF	∞	177	181	183	177	181	182	-	5	31	3
	Γ^{EDA}	176	181	183	176	181	183	-	5	31	3
	Γ^{IM}	177	181	182	177	181	182	-	5	32	4
MSDC	∞	146	166	166	-	152	152	3	-	77	20
	Γ^{EDA}	143	163	163	-	143	143	4	-	141	23
	Γ^{IM}	143	163	163	-	143	143	4	-	141	23
Time limit: 60 minutes											
MvS	∞	174	182	182	172	182	182	2	3	30	4
	Γ^{EDA}	179	182	182	177	182	182	2	3	30	4
	Γ^{IM}	181	183	184	179	182	182	2	2	28	2
MvI	Γ^{EDA}	173	179	180	171	179	180	5	1	238	6
	Γ^{IM}	179	182	182	177	182	182	3	2	30	4
EDF	∞	178	181	183	178	181	183	-	5	31	3
	Γ^{EDA}	178	181	183	178	181	183	-	5	31	3
	Γ^{IM}	179	181	183	179	181	183	-	5	31	3
MSDC	∞	146	166	166	-	152	152	3	-	77	20
	Γ^{EDA}	143	163	163	-	143	143	4	-	141	23
	Γ^{IM}	143	163	163	-	143	143	4	-	141	23

Table 3.8: Quality metrics for the computed solution S , as used in Tables 3.6 to 3.7

Column	Description
$\Pi^*\square$	counts loops for which Π^S is <i>proven</i> to be optimal by the approach, as discussed in Section 3.2.1
Π^*	counts loops for which Π^S is equal to the <i>optimal</i> Π
Π^\bullet	counts loops for which Π^S is equal to the <i>best-known</i> Π across all experiments, including loops for which $\Pi^\bullet = \Pi^*$
$T^*\square$	counts loops for which the Π and the schedule length are <i>proven</i> to be optimal by the approach
T^*	counts loops for which $\Pi^S = \Pi^*$, and T^S is equal to the <i>optimal</i> schedule length
T^\bullet	counts loops for which $\Pi^S = \Pi^\bullet$, and the schedule length T^S is equal to the <i>best-known</i> length, including loops for which these values are known to be optimal
no S	counts loops for which no solution could be computed within the time and attempt limits
OOM	counts loops for which no solution could be computed within the memory limit
$\Pi^S - \Pi^\bullet$	considers the differences between each computed and the loop's best-known Π . We set $\Pi^S = \Pi^{\max}$ in case no solution could be computed, and $\Pi^\bullet = \Pi^{\max}$ in case no solution is known (2 loops)
Σ	accumulates the differences
$\#$	shows the number of loops that have $\Pi^S > \Pi^\bullet$

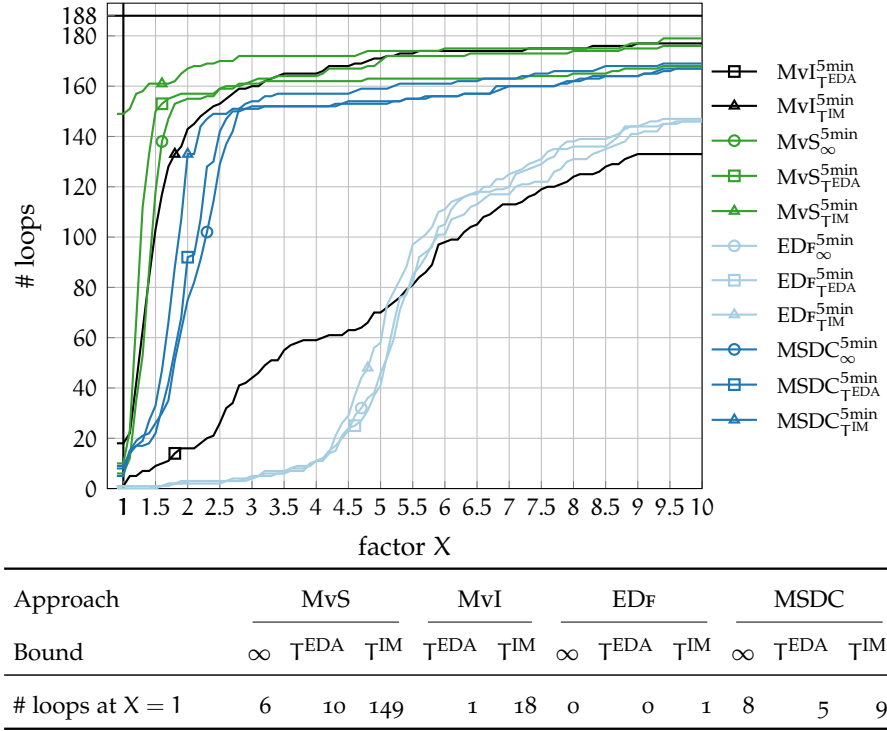


Figure 3.5: Performance profile of the scheduling times (5 minute time limit), showing the number of loops for which the scheduling time with a particular configuration is at most X times slower than the fastest scheduling time for each individual loop. The table shows the values for the special case $X = 1$, i.e. the number of loops for which a configuration defined the fastest scheduler runtime.

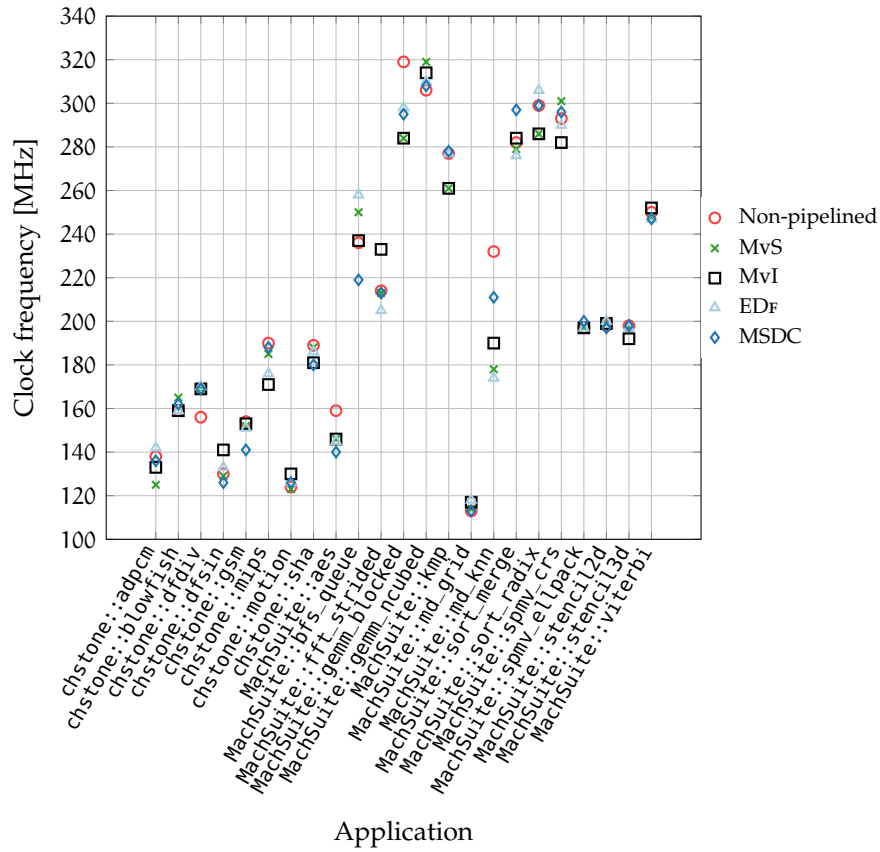


Figure 3.6: Maximum clock frequencies on Virtex-7 after HLS and place & route for the different schedulers (T^{IM} , 5 min configurations). Note that the Y-axis starts at 100 MHz.

3.3.6 FPGA Implementation

We used Nymble [42] to generate accelerator modules that are pipelined according to the schedules computed in the experiments with the T^{IM} bound and 5 minute time limit. The non-loop parts of the applications, as well as loops for which a particular scheduler could not compute a feasible modulo schedule, are not pipelined and rely on the non-modulo schedules computed by the heuristic fallback scheduler.

TaPaSCo [49] was employed to perform an out-of-context evaluation of these accelerators using Vivado 2018.2 targeting a Virtex-7 xc7vx690tffg1761-2 device. The target frequency was set to 200 MHz for the CHStone applications, MachSuite::aes and MachSuite::md_grid, and to 320 MHz for the remaining MachSuite applications.

Figure 3.6 shows the maximum clock frequencies when using the modulo schedulers, in comparison to a entirely non-pipelined accelerator using only the fallback non-modulo schedule for all loops. Unfortunately, chstone::aes and chstone::jpeg could not be implemented on the target device due to routing congestion in the Nymble-generated microarchitecture for any scheduler configuration, and are therefore excluded in the plot.

Overall, the frequency variations when using the different schedulers are moderate, and none of the approaches (especially not the non-pipelined one) dominates this experiment. However, note that the scheduling step is so early in the HLS flow that it cannot directly influence the later decisions made by the logic synthesis tool that ultimately determine the design's cycle time, so this result is not unexpected.

3.4 CHAPTER SUMMARY

In this chapter, we presented MOOVAC, an exact bi-criteria formulation of the Modulo Scheduling Problem that integrates the actual II minimisation and can be solved optimally by a standard ILP solver.

An extensive experimental study in the context of a high-level synthesis compiler showed that combining the Moovac formulation with a short time limit of 1 or 5 minutes per candidate II, and an improved upper bound for the schedule length, results in a practically usable modulo scheduling approach that finds optimal IIs in over 95 % of the test instances, and near-optimal IIs otherwise.

Exact modulo scheduling is justified by the clearly higher quality of the solutions compared to a state-of-the-art heuristic approach. In the MOOVAC formulation, modelling the central modulo-operator-constraints with overlap variables leads to overall shorter solver runtimes compared to a prior, well-tuned ILP formulation. Often, our approach is even faster than the heuristic SDC-based scheduler.

DEPENDENCE GRAPH PREPROCESSING FOR FASTER EXACT MODULO SCHEDULING

In Chapter 3, the EDFORM, which originates from a VLIW compiler context, performed worse on average than our MOOVAC-S formulation on our set of MSP instances taken from an HLS tool flow. However, given that the EDFORM has received widespread recognition in the community [28], we suspect that the slower performance may be partly due to the characteristics of HLS MSPs.

In this chapter, we present a compression algorithm for HLS MSP instances that yields problem structures which are more similar in size and density to instances expected by a VLIW modulo scheduler. Our work specifically targets *exact* modulo schedulers that are able to compute provably optimal solutions regarding a schedule length minimisation objective. The proposed algorithm retains this property. Applied to the EDFORM, we indeed show a mean speed-up of 4.37x for 21 large instances, which makes it competitive again with the HLS-tailored MOOVAC-S formulation.

This chapter is based on:

- [66] Julian Oppermann, Melanie Reuter-Oppermann, Lukas Sommer, Oliver Sinnen and Andreas Koch. ‘Dependence Graph Preprocessing for Faster Exact Modulo Scheduling in High-Level Synthesis’. In: *International Conference on Field Programmable Logic and Applications, FPL Dublin, Ireland*. 2018

4.1 ANALYSIS

Let us first revisit the relevant HLS and modulo scheduling foundations from Chapter 2.

4.1.1 Instances

Compared to the VLIW context, MSP instances that arise in an HLS flow are much larger, due to implicit loop-wide if-conversion, and have denser dependence graphs, due to the additional edges required to model operator chaining as well as to retain a sequential ordering of the loop’s memory accesses where needed.

This is apparent in our test MSP set (cf. Section 4.3): The 21 instances contain 471 operations and 1578 edges (median values). The ratio of the aforementioned additional, non-data-dependence edges ranges from 29% to 90% with a median value of 63%.

VLIW processors provide only a handful of functional units, and also need to limit access to architectural features such as register files and buses, resulting in tight constraints for all operations. In contrast, HLS generally aims to create spatially distributed computations (employing a dedicated hardware operator for each operation). This is possible, as many resources, e.g. LUTs, exist in abundance. Only a fraction of the operations needs to be time-multiplexed onto scarce shared operators, e.g. those performing floating-point arithmetic, or providing access to an off-chip memory. The spatial approach thus results in far fewer operator constraints that need to be included in the HLS MSP than in the VLIW MSP. In our test instances, for example, the share of operations using a shared operator type ranges between 3% and 29% with a median value of 12%.

4.1.2 Exact Schedulers

Exact modulo schedulers, in contrast to algorithms that rely on *heuristic* simplifications (e.g. [7, 40, 56]), model all the aspects of the MSP, and are therefore able to find provably optimal schedules. They are often defined in terms of a mathematical framework such as integer linear programs (e.g. [20, 26, 63]), or constraint programming (e.g. [6]), and, unfortunately, inherit the frameworks' exponential worst-case runtimes to find a solution. Still, they are a viable option in the context of an HLS system, as logic synthesis and place-and-route often requires multiple hours even on mid-sized FPGA devices anyway. This easily amortises the time spent to determine a high quality (provably optimal) solution by exact modulo scheduling.

However, recall from Section 2.7 that most exact modulo schedulers were proposed for and evaluated with VLIW-style MSP instances. Using the EDFORM as the representative for this class of schedulers, we showed in Chapter 3 that HLS-style instances require special attention in the design of a practical exact scheduler. Our Moovac-S formulation copes well with rather large instances, as the (many) unlimited operations are represented by a single integer decision variable. Similarly, a large number of dependence edges is unproblematic, as each edge results in only one linear constraint ((3.2) in Section 3.1).

In contrast, the EDFORM uses Π -many binary decision variables per operation, regardless whether the operation is subject to operator constraints or not. The authors achieved a significant speed-up by using Π -many, but 0-1-structured constraints to represent dependence edges ((2.15) in Section 2.8.1). These scalability difficulties when scheduling large and dense dependence graphs, and HLS-typical candidate Π s, make the EDFORM the ideal subject to demonstrate the strength of our proposed approach.

However, we expect other exact VLIW modulo schedulers, such as the formulations by Dinechin [20] and by Ayala and Artigues [4], to

benefit from the proposed problem compression as well. A common feature in these formulations is the use of *time-indexed* binary decision variables to model the operations' start times. This design typically results in complex linear constraints involving sums of subsets of these variables. Reducing the number of operations in the problem, and reducing the density of dependence edges therefore gives ample opportunity for speed-up when retrofitting these formulations for use in an HLS setting.

4.1.3 Critical Operations

The key insight for our proposed compression algorithm is that only some operations in an MSP instance are *critical* to the exact modelling, e.g. operations using a shared operator type, while others are not. Non-critical operations can be scheduled in an as-soon-as-possible manner, and thus, subgraphs of non-critical operations may be replaced by a single edge with a delay equal to the minimum delay of the subgraph's longest path. The compression makes it tractable to discover additional, superfluous dependence edges. Figures 4.1 to 4.4 demonstrate the proposed transformation. After scheduling the compressed instance, the start times of the critical operations are fixed in the original problem, and start times for the remaining non-critical operations can be determined in polynomial time.

Problem compression (or reduction) in general has been explored in the Operations Research community to make challenging problems more tractable by excluding non-critical aspects (e.g. [5]). To the best of our knowledge, ours is the first use of the technique to speed-up modulo scheduling.

4.2 MODULO SCHEDULING WITH COMPRESSED PROBLEMS

The input to the problem-compressing modulo scheduling approach presented in this chapter is an *instance* \mathcal{I} of the MSP, conforming to the problem signature introduced in Section 2.4. We assume that all shared operator types $q \in Q^{\text{Sh}}$ are fully pipelined and thus have $b_q = 1$.

RUNNING EXAMPLE In our usual notation, the MSP instance in Figure 4.1 is defined by the set of operations $O = \{1, 2, \dots, 11\}$, the set of edges $E = \{(1 \rightarrow 2), (1 \rightarrow 3), \dots\}$, and a simple operator model (not shown in the figure) that encompasses one unlimited $Q^\infty = \{q^\infty\}$ and one shared type $Q^{\text{Sh}} = \{q^{\text{Sh}}\}$, which provides a single operator $a_{q^{\text{Sh}}} = 1$. We have $O^{q^{\text{Sh}}} = \{2, 6, 9\}$, and use a grey contour to distinguish these operations from the one using the unlimited operator type. The operation latency l_i is 1 for all $i \in O$. There are no edge delays, i.e. $L_{ij} = 0$ for all $(i \rightarrow j) \in E$, and only a single backedge $(10 \rightarrow 3)$ with $d_{10\ 3} = 1$ and a dashed line style. All other edges $(i \rightarrow j) \in E$ are

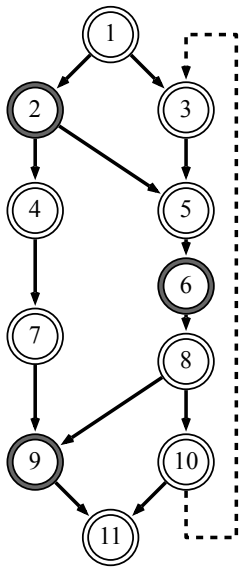


Figure 4.1: Example instance

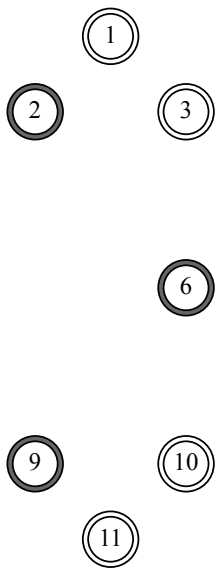


Figure 4.2: Compressed instance: critical operations

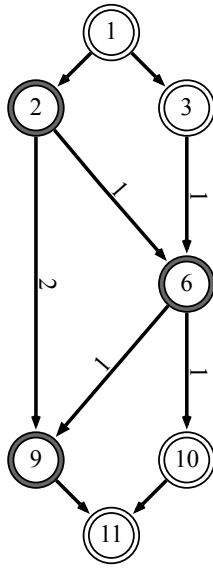


Figure 4.3: Compressed instance: constructed edges

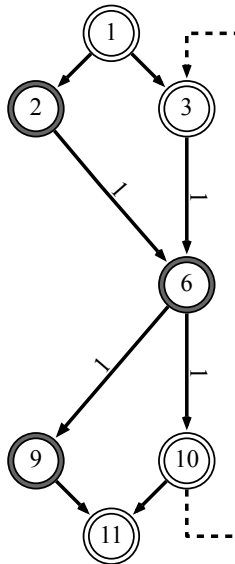


Figure 4.4: Compressed instance: edges filtered, backedges added

Table 4.1: Additional notation used in the problem-compressing approach

Symbol		Description
$\mathcal{J}, \mathcal{J}^{\text{Cmp}}$		Original and compressed instance of the MSP
O^{Cr}	$\subseteq O$	Critical operations
$\text{pred}(v)$	$\subseteq O, \forall v \in O$	Predecessors of operation v
$\text{succ}(v)$	$\subseteq O, \forall v \in O$	Successors of operation v
E^{Cr}	$\subseteq O^{\text{Cr}} \times O^{\text{Cr}}$	Edges between critical operations. Note that $E^{\text{Cr}} \not\subseteq E$
E^{Cons}	$\subseteq O^{\text{Cr}} \times O^{\text{Cr}}$	Edges constructed based on the result of the data-flow analysis
E^{Filt}	$\subseteq E^{\text{Cons}}$	Remaining edges after filtering transitively satisfied dependences
E^{BE}	$\subseteq E$	Backedges
$\text{LPL}_j^{\text{IN}}[k]$	$\in \mathbb{N}_0 \cup \{-\infty\}$	Incoming longest-path latency from preceding critical operation k to operation j
$\text{LPL}_j^{\text{OUT}}[k]$	$\in \mathbb{N}_0 \cup \{-\infty\}$	Outgoing longest-path latency from preceding critical operation k to operation j

forward edges with $d_{ij} = 0$. The lower bound for the interval Π^{\min} is 5, due to the recurrence spanned by the backedge. We assume a non-modulo scheduler would determine a schedule of length 7, and define Π^{\max} accordingly.

Our problem-compressing modulo scheduling approach comprises three steps. Table 4.1 summarises the additional notation used in the following sections.

First, given an HLS-style MSP instance \mathcal{J} , we derive a compressed problem instance \mathcal{J}^{Cmp} with the set of critical operations $O^{\text{Cr}} \subseteq O$ and infer a new set of edges $E^{\text{Cr}} \subseteq O^{\text{Cr}} \times O^{\text{Cr}}$. Then, \mathcal{J}^{Cmp} is scheduled with an arbitrary modulo scheduler, e.g. the EDFORM. The operation start times obtained from the solution to \mathcal{J}^{Cmp} are fixed in \mathcal{J} . This makes scheduling \mathcal{J} tractable, as it is no longer an operator-constrained scheduling problem and can thus be solved in polynomial time.

4.2.1 Construction of a Compressed Problem Instance

Let $E^{BE} = \{(i \rightarrow j) \in E : d_{ij} > 0\}$ be the set of all backedges in E , $\text{pred}(v) = \{i \mid \exists(i \rightarrow v) \in E \text{ with } d_{iv} = 0\}$ denote the set of predecessors of an operation v , and analogously, $\text{succ}(v) = \{j \mid \exists(v \rightarrow j) \in E \text{ with } d_{vj} = 0\}$ denote the set of successors of v .

CRITICAL OPERATIONS The proposed problem compression shall guarantee that a feasible/optimal solution to \mathcal{J}^{Cmp} can be completed to a feasible/optimal solution to \mathcal{J} .

The feasibility of \mathcal{J} for a candidate interval Π is subject to the interaction of the backedges (which impose deadlines, i.e. latest possible start times) and operator constraints (which may require operations to be scheduled in a later time step than required by their predecessors' finish times), as defined by (2.1) and (2.2). Additionally, operations without non-backedge predecessors must be included in O^{Cr} to capture the earliest start times in the schedule.

The schedule length depends, per definition, on the finish times (and in turn, on the start times) of operations without outgoing forward edges. These operations, in addition to the operations competing for shared operators, therefore need to be considered in \mathcal{J}^{Cmp} to ensure that an optimal solution for \mathcal{J}^{Cmp} may serve as the base for an optimal solution to \mathcal{J} .

With this consideration in mind, we define the set of *critical* operations O^{Cr} to contain

- all operations using a shared operator type, O^{Sh} ,
- operations at the endpoints of backedges, formally $\{v \mid \exists(i \rightarrow v) \in E^{BE} \vee \exists(v \rightarrow j) \in E^{BE}\}$, and
- operations with either no incoming or no outgoing forward edges, formally $\{v \mid \text{pred}(v) = \emptyset \vee \text{succ}(v) = \emptyset\}$.

Figure 4.2 depicts the set O^{Cr} for the running example: operations 2, 6 and 9 use the shared operator type, 3 and 10 are the endpoints to the only backedge, and 1 and 11 are the extremal vertices in the dependence graph.

In contrast, the start times of the remaining non-critical operations are determined only by the instance's forward edges when considering schedule length minimisation as the secondary objective. Entire subgraphs of non-critical operations can then be scheduled in an as-soon-as-possible manner between critical operations, given that enough time steps separate the critical operations. As a consequence, such subgraphs can be modelled in \mathcal{J}^{Cmp} by a single edge with the minimally required latency to fit the non-critical operations, as discussed in the next paragraphs.

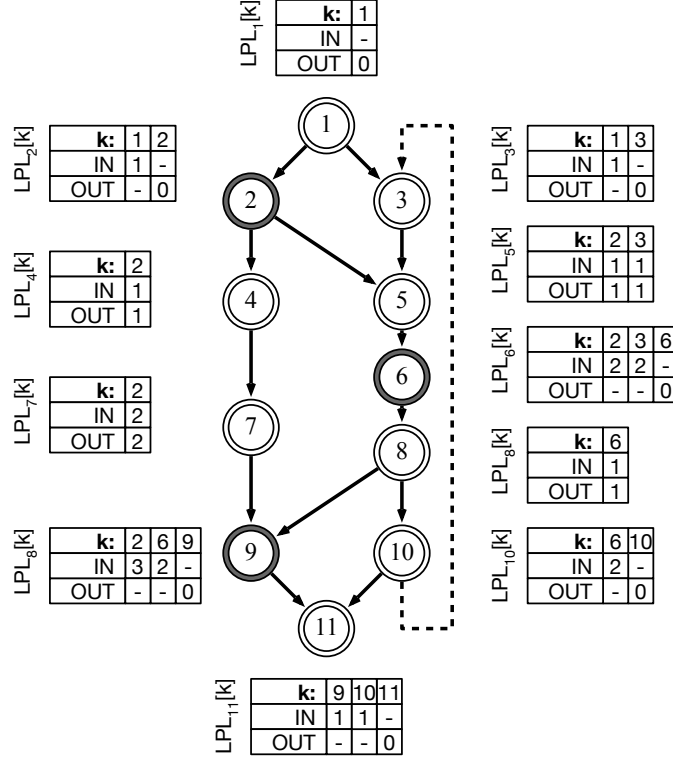


Figure 4.5: Longest path length analysis results for all $j \in O$: showing the values stored in $LPL_j^{IN}[k]$ and $LPL_j^{OUT}[k]$, for reachable preceding critical operations $k \in O^{Cr}$. Cells marked with “-” contain the value $-\infty$.

OVERVIEW OF EDGE CONSTRUCTION The set of edges E^{Cr} is not a subset of E . Rather, we begin with the construction of a new set of edges E^{Cons} , based on the computed length of the longest path in E from a critical operation to any other (critical or not) operation. Then, we filter out edges that are always satisfied, because a longer path exists between their endpoints, which yields the set E^{Filt} . Lastly, we include the backedges unchanged from the original instances. To summarise, we set

$$E^{Cr} = E^{Filt} \cup E^{BE}. \quad (4.1)$$

PATHS Let $(v_1 \rightsquigarrow v_h)$ denote a *path* from v_1 to v_h on the acyclic part of the dependence graph, i.e. a sequence $((v_1 \rightarrow v_2), (v_2 \rightarrow v_3), \dots, (v_{h-1} \rightarrow v_h))$ of pairwise distinct forward edges that connect a sequence of pairwise distinct operations (v_1, \dots, v_h) . We define the *length* of a path $(v_1 \rightsquigarrow v_h)$ in terms of the operation and edge latencies as $\sum_{x=1}^{h-1} l_{v_x v_{x+1}}$.

ANALYSIS We define a data-flow analysis to compute the length of the longest path from a critical operation $k \in O^{Cr}$ to an operation

$j \in O$. To that end, we associate each operation $j \in O$ with two dictionaries, $LPL_j^{IN}[k]$ and $LPL_j^{OUT}[k]$, which store the incoming and outgoing longest path length from a critical operation $k \in O^{Cr}$. The domain of the stored values is $\mathbb{N}_0 \cup \{-\infty\}$, where the value $-\infty$ is to be interpreted as “unreachable”. (4.2) and (4.3) are the corresponding data-flow equations.

$$LPL_j^{IN}[k] = \begin{cases} -\infty & \text{if } \text{pred}(j) = \emptyset \\ \max_{i \in \text{pred}(j)} LPL_i^{OUT}[k] + l_{ij} & \text{otherwise} \end{cases} \quad (4.2)$$

$$LPL_j^{OUT}[k] = \begin{cases} 0 & \text{if } j = k \\ -\infty & \text{if } j \neq k \wedge j \in O^{Cr} \\ LPL_j^{IN}[k] & \text{otherwise} \end{cases} \quad (4.3)$$

The important distinction from a standard longest-path computation is the fact that only paths from the *nearest preceding* critical operations are considered: for a critical operation $k \in O^{Cr}$, equation (4.3) sets the outgoing path length from itself ($j = k$) to 0, and resets the path length concerning all other critical operations to $-\infty$. For the running example, this happens i.a. at operation 9, as shown in Figure 4.5.

EDGE CONSTRUCTION After computing $LPL_j^{IN}[k]$, we construct new forward edges between the critical operations:

$$E^{Cons} = \{(u \rightarrow v), L_{uv} = LPL_v^{IN}[u] - l_u \mid \forall u, v \in O^{Cr} \text{ with } LPL_v^{IN}[u] > -\infty\} \quad (4.4)$$

Specifically, for all $v \in O^{Cr}$, we add an edge from the preceding critical operations $u \in O^{Cr}$, for which $LPL_v^{IN}[u] > -\infty$, to v . The corresponding edge latency L_{uv} is set to $LPL_v^{IN}[u] - l_u$. We subtract l_u , as otherwise we would account for u 's latency twice later.

Note that E^{Cons} is smaller than the transitive hull of E restricted to O^{Cr} , which would per definition connect all critical operations with each other. In contrast, in situations where all paths connecting two critical operations $u, v \in O^{Cr}$ contain at least one other critical operation, our construction scheme would not add the redundant edge $(u \rightarrow v)$.

Figure 4.3 shows E^{Cons} corresponding to the running example. Where specified, the edge labels now represent a non-zero edge latency, e.g. $L_{2 \ 9} = 2$. For all other edges, the edge latency remains 0.

EDGE FILTERING On the intermediate graph (O^{Cr}, E^{Cons}) , it is now tractable to compute all-pair-longest-path-length information $APLPL(u, v)$ for pairs of operations $u, v \in O^{Cr}$ using the Floyd-Warshall algorithm [32], which we use to filter out superfluous direct edges

whose implied precedence constraints are satisfied transitively by other edges: an edge $(u \rightarrow v) \in E^{\text{Cons}}$ is removed iff $L_{uv} < \text{APLPL}(u, v)$.

The final, compressed version of the running example is illustrated in Figure 4.4.

COMPLEXITY In a single pass over E , we precompute the sets E^{BE} , and $\text{pred}(i), \text{succ}(i)$ for all operations $i \in O$. Collecting all critical operations O^{Cr} requires visiting all operations in O once. As we only consider forward edges when computing $\text{LPL}_j^{\text{IN}}[k]$, the data-flow analysis operates on an acyclic graph and reaches its fix point after handling each operation $j \in O$ (which requires inspecting the $|\text{pred}(j)|$ predecessors) once in topological order. Performing all these preparation steps is in $\mathcal{O}(|O| + |E|)$.

Computing E^{Cons} has a worst-case runtime of $\mathcal{O}(|O^{\text{Cr}}|^2)$, while the edge filtering step is in $\mathcal{O}(|O^{\text{Cr}}|^3)$ due to the longest-path calculation. However, as HLS-style MSP instances are characterised by $|O^{\text{Cr}}| \ll |O|$, the resulting runtimes are negligible compared to the runtime of the actual modulo scheduler even for the cubic parts of the compression algorithm.

4.2.2 Modulo Scheduling

The compressed problem instance \mathcal{J}^{Cmp} comprises the sets O^{Cr} and E^{Cr} , and inherits all other parameters from the original instance \mathcal{J} . We obtain a solution S^{Cmp} with a feasible interval $\Pi^{S^{\text{Cmp}}}$ and start times $t_k^{S^{\text{Cmp}}}$ for all $k \in O^{\text{Cr}}$ from solving \mathcal{J}^{Cmp} with any modulo scheduler. In case the nested modulo scheduler does not find any solution to \mathcal{J}^{Cmp} , our approach will stop here and report the failure.

4.2.3 Schedule Completion

Lastly, start times for the non-critical operations $i \in O \setminus O^{\text{Cr}}$ have to be computed. To this end, we solve the following **ILP**, defined for integer variables t_i :

$$\min \quad T \tag{4.5}$$

$$\text{s.t.} \quad t_i + l_{ij} \leq t_j + d_{ij} \cdot \Pi^{S^{\text{Cmp}}} \quad \forall (i \rightarrow j) \in E \tag{4.6}$$

$$t_i = t_i^{S^{\text{Cmp}}} \quad \forall i \in O^{\text{Cr}} \tag{4.7}$$

$$t_i + l_i \leq T \quad \forall i \in O \tag{4.8}$$

We obtain start times t_i^S for all operations i from the corresponding decision variables in the ILP above. Together with $\Pi^S = \Pi^{S^{\text{Cmp}}}$, as computed in the previous step, this schedule yields a complete solution S to the original problem instance \mathcal{J} .

COMPLEXITY The ILP above can be easily transformed into an SDC [15]. SDCs comprise a special class of ILPs that are optimally solvable in polynomial time, due to the fact that their LP relaxation is guaranteed to produce an integer-valued solution.

4.3 EXPERIMENTAL EVALUATION

We evaluate our problem-compressing modulo scheduling approach on a set of realistic test instances: C applications from the HLS benchmark suites CHStone [38] and MachSuite [75] are compiled with Clang to LLVM-IR [53]. We use LLVM/Clang version 3.3, optimisation preset “-O2” with loop unrolling disabled, and perform exhaustive inlining. Nymble [42] constructs per-loop control-data-flow graphs (CDFG), where the original control flow is replaced by multiplexers and predicated operations. Nested loops become special operations in the graph, thus making it possible to modulo schedule all loops in the application instead of being limited to the most deeply nested ones. Nymble also constructs edges to retain the sequential order amongst memory access operations where needed, as determined by LLVM’s dependence analysis. We use operator latencies and physical delays from the Bambu HLS framework’s [71] extensive operator library for a Xilinx xc7vx690 device. For each operator, we choose the lowest-latency variant that is estimated to achieve a frequency of at least 250 MHz. The edges to limit operator chaining are constructed with a simple path-based approach similar to [43] and aim to enforce a maximum cycle time of 5 ns.

From a total set of 354 loops, we selected 21 unique instances that took more than 10 seconds to schedule with at least one scheduler configuration. The MSP instances use the following allocation of shared operator types: memory load (2 available)/store (1), nested loop (1), integer division (8), floating-point addition (4), FP subtraction (4), FP multiplication (4), other FP operations (2 each).

In the second step (Section 4.2.2) of our approach, we delegate the actual modulo scheduling to the ED_{FORM} as in Section 2.8.1, and to the MOOVAC-S formulation as in Section 3.1.1. Both formulations minimise the schedule length.

We use CPLEX 12.6.3 to solve the MOOVAC-S and ED_{FORM} ILPs, as well as for the ILP used in the schedule construction step (Section 4.2.3). We set a time limit λ of 3 minutes (minus the time to construct the linear program via the solver’s API) per scheduling attempt/candidate Π , and try at most 20 candidate Π s per instance, thereby capping its total runtime to one hour. The solver uses deterministic multithreading with up to 8 threads. We run up to two scheduling jobs concurrently [88] on 2x12-core Intel Xeon E5-2680 v3 systems running at 2.8 GHz with 64 GB RAM. The scheduler runtimes presented here correspond

Note that this is not 4 ns/250 MHz, to give some headroom for the rest of the synthesis flow.

to the best (in terms of the computed Π , the schedule length, and lastly, the scheduler runtime) of three runs.

Table 4.2 shows the effects of our algorithm to the input problem size regarding the signature of the MSP. The instances are named according to the CHStone/MachSuite benchmark application they originated from, and distinguished by an arbitrary loop ID assigned by the Nymble compiler. The size of the original instance \mathcal{I} is specified by the number of operations $|\mathcal{O}|$, edges $|\mathcal{E}|$, operations using a shared operator type $|\mathcal{O}^{\text{Sh}}|$, and backedges $|\mathcal{E}^{\text{BE}}|$. Recall that $\mathcal{O}^{\text{Sh}} \subseteq \mathcal{O}$ and $\mathcal{E}^{\text{BE}} \subseteq \mathcal{E}$. For example, instance `aes2:2` has 155 operations, of which 34 use shared operator types, as well as 291 edges in total, of which 26 are backedges. The size of the compressed instance is specified accordingly by the number of critical operations $|\mathcal{O}^{\text{Cr}}|$, and inferred edges $|\mathcal{E}^{\text{Cr}}|$. By construction, the sets \mathcal{O}^{Sh} and \mathcal{E}^{BE} are carried over unchanged to the compressed problem, and therefore their sizes are not repeated in the table. In the next column, we report the time required for the problem compression, including all steps of our approach without the actual modulo scheduling with the delegate modulo scheduler.

Table 4.3 shows the effects of our problem compression to the size of the ILPs, i.e. the number of decision variables $\text{Vars}_{\mathcal{X}}$ and the number of linear constraints $\text{Cons}_{\mathcal{X}}$, when applying either the EDFORM or the MOOVAC-S formulation to an instance \mathcal{X} . We present the absolute numbers for the original instance \mathcal{I} , and relative to that, the quantities for the compressed instance \mathcal{I}^{Cmp} . For example, the ILP according to the EDFORM for `aes2:2` has roughly 3600 decision variables initially. After the problem compression, only 26% remain, i.e. the ILP for the compressed instance requires about 936 variables. Overall, EDFORM ILPs are compressed to 15% of variables and 22% of constraints (excluding `dfs in:1`), whereas MOOVAC-S ILPs retain over 83% of their original sizes (geometric means).

In Tables 4.4 to 4.5, overall runtimes to schedule the original instances with and without our proposed problem-compressing approach are shown, alongside the initiation intervals Π^{S} and the corresponding schedule lengths T^{S} of the respective computed solution S . Asterisks denote which parts of the solution were proven to be optimal. We distinguish four cases: “(*, *)” solutions are optimal according to both objectives, i.e. a minimum length schedule was found for the smallest feasible Π . In a “(*,)” solution, the Π^{S} is proven to be optimal, but the solver was not able to find or prove the optimal schedule length. This typically means that the solver depleted its 3 min time budget, and we accepted any feasible solution. We only know that we have a feasible solution “(,)” if at least one scheduling attempt was aborted due to the time limit without finding a solution, and was

As both suites contain an application named aes, we refer to MachSuite’s version as aes2.

As the number of variables and constraints in the EDFORM is dependent on the candidate Π , we present here the ILP size for the last (= successful) scheduling attempt.

Refer to Section 3.2 for a more detailed discussion on the different optimality results.

Table 4.2: Compression results: Problem size

Instance	J				J^{Comp}		time [s]
	$ O $	$ E $	$ O^{Sh} $	$ E^{BE} $	$ O^{Cr} $	$ E^{Cr} $	
fft_str:::2	84	290	24	65	29	112	0.02
aes2:::2	155	291	34	26	41	92	0.03
aes:::2	177	377	29	25	34	81	0.02
gsm:::3	194	313	18	12	43	120	0.03
aes2:::4	225	683	62	158	69	285	0.05
aes:::15	236	439	32	2	37	81	0.03
md_grid:::7	300	577	24	53	35	115	0.02
jpeg:::87	380	1574	35	102	40	171	0.04
jpeg:::63	382	1577	35	102	40	171	0.04
jpeg:::47	383	1578	35	102	40	171	0.04
jpeg:::59	471	1919	50	223	55	308	0.05
jpeg:::19	476	1609	69	363	74	489	0.07
jpeg:::41	480	1956	50	224	55	309	0.05
adpcm:::2	710	1478	100	82	105	388	0.12
adpcm:::1	777	1746	95	141	106	396	0.11
blowfish:::1	789	2558	107	42	118	517	0.12
jpeg:::17	942	5047	106	1041	111	1210	0.15
mips:::1	1076	4441	65	1155	76	1595	0.09
aes:::12	1367	3065	205	532	210	911	0.27
aes:::4	1374	2816	205	402	212	785	0.31
dfs:::1	2651	38642	67	1222	76	1402	0.10

Table 4.3: Compression results: ILP size

Instance	EDFORM				MOOVAC-S			
	Vars _J	$\frac{\text{Vars}_{J\text{Cmp}}}{\text{Vars}_J}$	Cons _J	$\frac{\text{Cons}_{J\text{Cmp}}}{\text{Cons}_J}$	Vars _J	$\frac{\text{Vars}_{J\text{Cmp}}}{\text{Vars}_J}$	Cons _J	$\frac{\text{Cons}_{J\text{Cmp}}}{\text{Cons}_J}$
fft_str:::2	6.5k	35%	22.1k	40%	0.4k	85%	1.1k	84%
aes2:::2	3.6k	26%	6.3k	32%	1.5k	92%	4.6k	96%
aes:::2	8.7k	19%	18.1k	22%	0.7k	79%	1.9k	84%
gsm:::3	2.3k	22%	3.4k	36%	0.5k	72%	1.3k	86%
aes2:::4	7.9k	31%	22.6k	42%	3.8k	96%	12.4k	97%
aes:::15	3.5k	16%	6.0k	18%	1.4k	85%	4.1k	91%
md_grid:::7	14.4k	12%	27.0k	20%	1.1k	75%	3.1k	85%
jpeg:::87	15.6k	11%	61.1k	11%	1.6k	78%	5.4k	74%
jpeg:::63	15.7k	10%	61.2k	11%	1.6k	78%	5.4k	74%
jpeg:::47	15.7k	10%	61.2k	11%	1.6k	78%	5.4k	74%
jpeg:::59	23.6k	12%	91.8k	16%	2.4k	83%	8.2k	80%
jpeg:::19	26.2k	16%	85.3k	30%	4.8k	92%	16.2k	93%
jpeg:::41	24.5k	11%	95.5k	16%	2.4k	82%	8.3k	80%
adpcm:::2	37.6k	15%	76.2k	26%	9.3k	93%	30.6k	96%
adpcm:::1	50.5k	13%	110.8k	21%	8.4k	92%	27.5k	95%
blowf:::1	90.7k	15%	289.1k	20%	18.6k	96%	63.8k	97%
jpeg:::17	105.5k	11%	553.3k	23%	9.5k	91%	33.9k	89%
mips:::1	34.4k	6%	132.1k	32%	5.7k	82%	19.9k	86%
aes:::12	128.5k	15%	283.3k	29%	45.8k	97%	156.5k	99%
aes:::4	129.2k	15%	260.7k	28%	46.3k	97%	158.0k	99%
dfs:::1	†	16.7k	†	305.3k	6.7k	62%	52.3k	29%

† Timeout during ILP construction for original instance, showing absolute numbers for compressed instance.

Table 4.4: Scheduling results: ED_{FORM}

Instance	J			jCmp			speed-up
	time [s]	Π^S	τ^S	time [s]	Π^S	τ^S	
fft_strided::2	45.04	*74	*77	13.04	*74	*77	3.45
aes2::2	26.77	*20	*21	4.70	*20	*21	5.69
aes::2	18.60	*46	*48	2.92	*46	*48	6.37
gsm::3	0.45	*9	*20	0.25	*9	*20	1.82
aes2::4	1277.49	32	33	703.53	32	33	1.82
aes::15	2.04	*12	*18	0.29	*12	*18	6.99
md_grid::7	15.80	*45	*45	2.20	*45	*45	7.19
jpeg::87	16.26	*38	*47	1.22	*38	*47	13.33
jpeg::63	16.33	*38	*47	1.22	*38	*47	13.41
jpeg::47	15.88	*38	*47	1.27	*38	*47	12.51
jpeg::59	42.08	*47	*55	3.12	*47	*55	13.47
jpeg::19	180.00	*52	61	18.27	*52	*54	9.85
jpeg::41	48.20	*48	*56	3.40	*48	*56	14.17
adpcm::2	180.00	*50	132	39.59	*50	*85	4.55
adpcm::1	1620.00	62	94	758.32	58	72	2.14
blowfish::1	3600.00	-	-	3600.12	-	-	1.00
jpeg::17	1080.00	-	-	86.08	*104	*105	12.55
mips::1	1077.93	29	34	398.66	26	38	2.70
aes::12	3600.00	-	-	3600.27	-	-	1.00
aes::4	3600.00	-	-	3600.31	91	92	1.00
dfs::1	3240.00	-	-	3240.10	-	-	1.00
sum [min]	328.38			267.98			
geomean							4.37

Asterisks (*) indicate optimality was proven

Instance	\mathcal{J}			\mathcal{J}^{Cmp}			speed-up
	time [s]	Π^S	Γ^S	time [s]	Π^S	Γ^S	
fft_strided::2	0.54	*74	*77	0.80	*74	*77	0.67
aes2::2	37.24	*20	*21	27.00	*20	*21	1.38
aes::2	2.97	*46	*48	3.35	*46	*48	0.89
gsm::3	36.34	*9	*20	180.03	*9	20	0.20
aes2::4	2520.00	33	33	2700.04	34	33	0.93
aes::15	180.00	*12	18	180.02	*12	18	1.00
md_grid::7	6.20	*45	*45	5.75	*45	*45	1.08
jpeg::87	0.28	*38	*47	0.48	*38	*47	0.59
jpeg::63	0.25	*38	*47	0.48	*38	*47	0.52
jpeg::47	0.28	*38	*47	0.48	*38	*47	0.58
jpeg::59	1.01	*47	*55	1.10	*47	*55	0.92
jpeg::19	1.53	*52	*54	4.65	*52	*54	0.33
jpeg::41	1.05	*48	*56	1.21	*48	*56	0.87
adpcm::2	180.00	*50	94	180.10	*50	92	1.00
adpcm::1	1440.00	61	81	1080.11	59	73	1.33
blowfish::1	3600.00	-	-	3571.82	-	-	1.01
jpeg::17	8.93	*104	*105	11.47	*104	*105	0.78
mips::1	2160.00	35	34	1980.08	34	34	1.09
aes::12	3600.00	-	-	3600.29	-	-	1.00
aes::4	3600.00	-	-	3600.27	-	-	1.00
dfs::1	24.28	*201	*216	24.89	*201	*216	0.98
sum [min]	290.02			285.91			
geomean							0.80

Asterisks (*) indicate optimality was proven

conservatively classified as infeasible. In the worst case, no solution was found for any candidate II, for which we note “-” in the table.

The results clearly show that our problem-compressing modulo scheduling approach significantly speeds-up scheduling with the EDFORM. The accumulated runtime to schedule the 21 test instances is improved by 18%. The per-instance speed-ups are never below 1x, and range up to 14x, with a geometric mean of 4.37x. As a side effect, the solution quality for the largest instances is improved as well: the solver is able to turn feasible solutions into optimal ones (e.g. jpeg : 19), or finding feasible solutions at all (e.g. jpeg : 17). Compared to the solver runtimes, the time spent for the problem compression and schedule completion is negligible.

The distribution of the IIs in our set of test instances (cf. Table 4.4) plays an important role in explaining these substantial performance gains. Considering that each operation in the EDFORM is represented by II-many binary variables, and each edge is expressed by II-many linear constraints, it is obvious (and evident in Table 4.3) that any reduction in the input problem size will have a huge impact on the size of the constructed and solved ILP.

The accumulated runtime with the MOOVAC-S formulation improves marginally, but the per-instance speed-ups ranging from 0.2x to 1.33x indicate that MOOVAC-S is not amenable to be accelerated by our problem-compressing approach. We attribute the small positive gains to the already efficient modelling of non-critical operations and edges in the MOOVAC-S formulation, as documented in Table 4.3. Most of the performance regressions occur on compressed instances that are already scheduled in very short runtimes in their original versions, and can be considered negligible in practice. However, instances gsm : 3 and aes2 : 4 incur a loss of solution quality. We believe that this is caused by a phenomenon called *performance variability* which is inherent to ILP solvers. In a nutshell, even small structural changes to linear programs may significantly influence the solver runtime [57]. As removing redundant constraints and decision variables is at the core of our approach, it is susceptible to this effect.

4.4 CHAPTER SUMMARY

We presented a problem-compression algorithm intended to be used in conjunction with exact modulo schedulers whose internal structure does not scale well with the size and density of dependence graphs typical for HLS-style MSP instances. Applied to the ILP formulation by Eichenberger and Davidson, we were able to unconditionally speed up all instances in our benchmark set, and obtain better quality solutions for the largest loops.

With our new preprocessing of the dependence graphs, the EDFORM and MOOVAC-S formulations are much closer performance-wise on

HLS-typical MSP instances. However, the two schedulers still exhibit different strengths and weaknesses, which make each of them better suited for a certain set of MSP instances.

DESIGN-SPACE EXPLORATION WITH MULTI-OBJECTIVE, RESOURCE-AWARE MODULO SCHEDULING

Up until now, we considered the allocation of operators to be a parameter to the **MSP**. This was a valid assumption, because even though **HLS** tools leverage the reconfigurable nature of **FPGAs** for application-specific allocations, they usually determine the number of operators *independent* of the scheduling phase, despite the fact that both phases are highly intertwined [35].

To that end, we make the following contributions in this chapter. First, we present an extension to the formal problem signature of the **MSP** from Section 2.4 to make it *resource-aware*, in the sense that the operator allocation is variable, computed during the actual modulo scheduling, and only subject to low-level resources available on the **FPGA** device. We adapt the bounds to the **II** search space to the new situation, and introduce analogous bounds for the space of possible allocations. Secondly, we discuss how to efficiently compute all Pareto-optimal trade-off solutions for such an extended problem. After experimenting with a standard method from multi-criteria optimisation, we propose a novel problem-specific approach, which, according to our evaluation, outperforms the standard method in terms of both overall runtime and number of trade-off points.

This chapter is based on:

- [67] Julian Oppermann, Patrick Sittel, Martin Kumm, Melanie Reuter-Oppermann, Andreas Koch and Oliver Sinnen. ‘Design-Space Exploration with Multi-Objective Resource-Aware Modulo Scheduling’. In: *International Conference on Parallel and Distributed Computing, Euro-Par, Göttingen, Germany*. 2019

5.1 SCHEDULING FRAMEWORK

Figure 5.1 illustrates the kind of trade-offs we want to explore in this chapter. Assume we have two shared operator types, **ADD** and **MUL**. On the left, we show the fully-spatial solution, in which an operator instance is allocated for every operation. This is required to achieve the best-possible **II** of 1, resulting in the best throughput. On the right, the smallest possible allocation is shown, providing only a single instance of each operator type. As a consequence, we can only execute this solution with an **II** of 4. Lastly, in the middle, we show a compromise between the two extremes, with an **II** of 2 and two allocated operators per type.

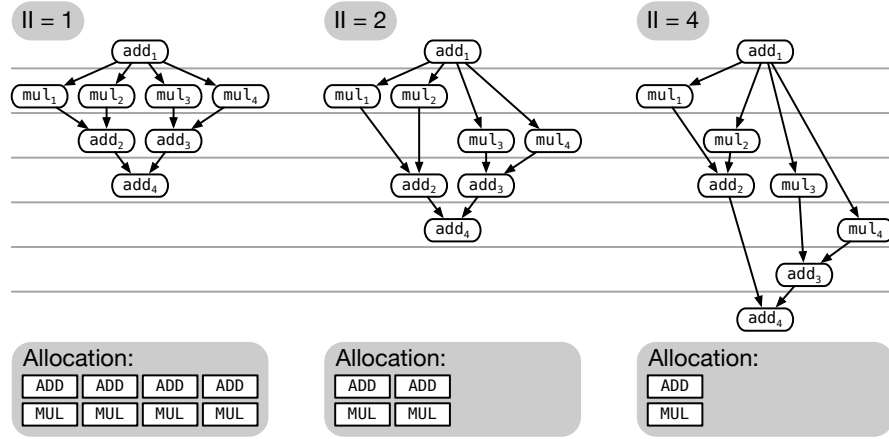


Figure 5.1: Different trade-offs regarding the throughput (smaller II is better) and resource demand (fewer allocated operators is better)

In this simple example, we assume that the ADD and MUL operators have the same resource demand, and therefore show only the number of allocated operators. In reality, though, different operator types require different subsets of the the FPGA's resources. To that end, we have to model the demand for, and the usage of the resources as part of the optimisation problem.

This chapter builds upon the work of Šůcha and Hanzálek [86] to define a theoretical framework in which the intertwined HLS (modulo) scheduling and allocation problems are modelled together, and presents methods to compute all Pareto-optimal trade-off solutions. We do not attempt to assign a practical value to each solution, because that depends entirely upon what a designer wants to achieve in a particular situation.

5.1.1 The Resource-Aware Modulo Scheduling Problem

The Resource-Aware Modulo Scheduling (RAMS) problem is an extension of the MSP as introduced in Section 2.4. Table 5.1 summarises the proposed modifications to the problem signature.

The FPGA is abstracted to the low-level resource types R it provides. We have N_r elements of each resource type $r \in R$ available for the allocation of operators. The operator types are additionally characterised by their resource demands: Each q -instance, $q \in Q$, occupies n_q^r elements of resource type $r \in R$.

The most important modification, compared to the MSP, is that the allocation is no longer an input parameter to the problem, but returned as part of a solution S , in form of the individual number of instances a_q^S of operator type $q \in Q$.

Table 5.2 summarises supplementary notations used in this chapter. For conciseness, we introduce (5.1) as an alternative notation to rep-

These limits can coincide with the amount of resources that are physically available on the target FPGA, but are not required to (cf. Chapter 6).

Table 5.1: Problem signature modification for the Resource-Aware Modulo Scheduling problem

Symbol	Description
INPUT:	
R	Resource types
$N_r \in \mathbb{N}_0, \forall r \in R$	Available number of elements of resource type r
$n_q^r \in \mathbb{N}_0, \forall q \in Q, \forall r \in R$	Resource demand of one instance of operator type q , regarding resource type r
$a_q \in \mathbb{N}, \forall q \in Q^{\text{Sh}}$	Allocation (now an output)
OUTPUT:	
$a_q^S \in \mathbb{N}_0, \forall q \in Q$	Allocation, i.e. a number of instantiated operators for each type q

Table 5.2: Supplementary notations

Symbol	Description
$A = \langle a_{q_1}, \dots, a_{q_{ Q }} \rangle$	Alternative notation for an allocation, grouping together the individual allocations for all operator types
$n^r(A) \in \mathbb{N}_0, \forall r \in R$	Accumulated resource demand of an allocation A
A^{\min}, A^{\max}	Minimum, maximum allocation
$A^{\text{tr}}(\Pi^S)$	Trivial allocation for Π^S
$f^{\Pi}(S) \in \mathbb{N}$	Objective function for the Π minimisation
$f^{\text{RU}}(S) \in [0, 1] \subset \mathbb{R}$	Objective function for the resource utilisation minimisation
\mathcal{S}	Set of Pareto-optimal solutions for an instance of the RAMS problem
$X = \langle \Pi^X, A^X, \dots \rangle$	“Special”, intermediate solution – denotes the (still variable) components of a RAMS solution within an ILP-based modulo scheduling formulation

We also consider unlimited operator types in the allocation, because even though we know how many instances we need, we still have to account for the resources they occupy.

represent an allocation, i.e. as a vector grouping together the individual numbers of instances regarding all operator types.

$$A = \langle a_{q_1}, \dots, a_{q_{|Q|}} \rangle \quad (\text{assuming } Q = \{q_1, \dots, q_{|Q|}\}) \quad (5.1)$$

For example, A^S denotes the allocation associated with a solution S . (5.2) defines the accumulated resource demand of an allocation.

$$n^r(A) = \sum_{q \in Q} a_q \cdot n_q^r \quad (5.2)$$

The new *resource constraints* (5.3) express that the accumulated resource demand resulting from the allocation of a solution S must not exceed the given resource limits.

$$n^r(A^S) \leq N_r \quad \forall r \in R \quad (5.3)$$

5.1.2 The Multi-Objective RAMS Problem

As motivated at the beginning of this chapter, two competing minimisation objectives exist in our setting. The first objective function, $f^{\Pi}(S)$ defined in (5.4), simply extracts the Π from a solution S . The second objective function $f^{\text{RU}}(S)$ models the *resource utilisation* of S . As shown in (5.5), we sum up the accumulated resource demands for each resource type, weighted by the resource's relative scarcity. The value is then scaled by $1/|R|$ in order to get a real value between 0 and 1.

$$f^{\Pi}(S) = \Pi^S \quad (5.4)$$

$$f^{\text{RU}}(S) = \frac{1}{|R|} \sum_{r \in R} \frac{n^r(A^S)}{N_r} \quad (5.5)$$

The benefit of this definition is that almost any change in the operator allocation is reflected in the value of the objective function, whereas considering only the maximum relative utilisation among the resource types ($\max_{r \in R} \frac{n^r(A^S)}{N_r}$) would potentially map many different allocations to the same resource objective value.

The existence of different solutions with the same objective value makes it harder for the ILP solver to prove the optimality of a solution.

Less formally, a solution is Pareto-optimal if no solution exists that is better in terms of both objectives.

In contrast to the base MSP, no universally applicable order exists for these objectives. Instead, we seek to compute a set \mathcal{S} of *Pareto-optimal* solutions with different trade-offs between the two objectives, and refer to this endeavour as the Multi-Objective Resource-Aware Modulo Scheduling (**MORAMS**) problem.

A solution $S \in \mathcal{S}$ is Pareto-optimal if it is *not dominated* by any other solution $X \in \mathcal{S}$, i.e. $\nexists S' \in \mathcal{S}$ with $(f^{\Pi}(S'), f^{\text{RU}}(S')) < (f^{\Pi}(S), f^{\text{RU}}(S))$.

LATENCY Note that we do not consider the schedule length as a third objective here. The minimisation of the Π and the minimisation of the schedule length both conflict with the resource utilisation objective.

However, similar as in the traditional MSP, we rather want to “invest” resources in operators that enable the loop to achieve a lower Π , instead of a shorter schedule length, though allocating more operators to achieve a smaller Π may also benefit the schedule length, as evident in Figure 5.1. Still, the schedule length should also not be completely unconstrained, as a an accelerator design with a very long latency would waste resources in places opaque to the scheduling problem, e.g. in the controller circuit, or for registers storing intermediate values for long periods of time. To that end, we will experiment with different external latency constraints in Section 5.4, modelled as different values for T^{\max} , the upper bound for the schedule length.

5.1.3 Bounds

The solution space for the MORAMS problem can be confined by simple bounds derived from the problem instance.

ALLOCATION We introduce the minimum allocation A^{\min} and the maximum allocation A^{\max} , comprised of conservative lower and upper bounds for the per-operator type allocation of any solution S , $a_q^{\min} \leq a_q^S \leq a_q^{\max}$ for each $q \in Q$. Note that both bounds may be infeasible for a given RAMS problem instance, as A^{\min} may provide too few operators to obey temporal deadlines, and A^{\max} may use too many resources to fit within the given resource limits.

Per definition, we use as many instances of unlimited operator types as needed. To reflect that, we fix their allocation to the number of operations using them (5.6).

$$|O^{q^\infty}| = a_{q^\infty}^{\min} = a_{q^\infty}^S = a_{q^\infty}^{\max} = |O^{q^\infty}| \quad \forall q^\infty \in Q^\infty \quad (5.6)$$

For the remaining shared operator types Q^{Sh} , we propose the definitions in (5.7) and (5.8). The minimum allocation A^{\min} provides one instance per type. We assume that $n^r(A^{\min}) \leq N_r$, regarding all resources r , as otherwise the problem instance is trivially infeasible. The maximum allocation A^{\max} models how many operators of a particular type would fit on the device if all other operator types were fixed at their minimum allocation.

$$\forall q \in Q^{\text{Sh}} : \quad a_q^{\min} = 1 \quad (5.7)$$

$$a_q^{\max} = \min \left\{ \underbrace{1}_{(a)} + \underbrace{\min_{r \in R: n_q^r > 0} \left\lfloor \frac{N_r - n^r(A^{\min})}{n_q^r} \right\rfloor}_{(b)}, \underbrace{|O^q|}_{(c)} \right\} \quad (5.8)$$

Here, (a) represents the one q -instance already considered in the minimum allocation, (b) models how many extra q -instances would fit using the remaining elements of resource r , i.e. when subtracting the

accumulated r -demand of the minimum allocation. Lastly, (c) limits the allocation to its trivial upper bound, i.e. the number of operations that use q .

INITIATION INTERVAL The adaption of the static bounds for the Π search space is straight-forward: For the computation of the lower bound Π^{\min} , we plug in the maximum allocation A^{\max} , and analogously, the computation of the upper bound Π^{\max} is subject to the minimum allocation A^{\min} .

INTERACTION OF ALLOCATION AND Π In an instance of the RAMS problem, the allocation and the Π are not independent. Recall from Section 2.6 that we used the pigeonhole principle to derive a lower bound Π^{opr} from the allocation, which was a parameter then. The main insight was that each operator provides only a limited Π - and blocking-time-dependent number of slots to bind operations to. The same line of thought still applies when both the Π and the allocation are variables in some solution S . We obtain the lower bounds (5.9) and (5.10), which can already be employed while a solution is computed.

$$\Pi^S \geq \left\lceil \frac{|O^q| \cdot b_q}{a_q^S} \right\rceil \quad \forall q \in Q^{\text{Sh}} \quad (5.9)$$

$$a_q^S \geq \left\lceil \frac{|O^q| \cdot b_q}{\Pi^S} \right\rceil \quad \forall q \in Q^{\text{Sh}} \quad (5.10)$$

Other than the change in notation, the allocation-dependent lower bound for the Π (5.9) is equivalent to (2.7). Inequality (5.10) is the Π -dependent lower bound for the allocation, and results from simply exchanging the Π and the allocation in (5.9).

5.1.4 Trivial Allocation

The smallest-possible Π -dependent allocation will play an important role later in the discussion of the approaches to solve the MORAMS problem. We call it *trivial allocation* $A^{\text{tr}}(\Pi^S)$ for an interval Π^S . The definition in (5.11) sets the individual allocations for every shared operator type to be equal to the right-hand side of (5.10).

$$a_q^{\text{tr}}(\Pi^S) = \left\lceil \frac{|O^q| \cdot b_q}{\Pi^S} \right\rceil \quad \forall q \in Q^{\text{Sh}} \quad (5.11)$$

5.2 EXTENSION OF EXISTING ILP FORMULATIONS

The template formulation in Figure 5.2 illustrates how ILP-based modulo scheduling formulations can be made resource-aware with small changes. In principle, it suffices to replace formerly constant limits

$$\begin{aligned}
& \text{minimise} && f^{\text{RU}}(X) \\
& \text{subject to} && \text{formulation-specific dependence constraints} \quad (\rightarrow 2.1) \\
& && \text{formulation-specific constraints that ensure at most } a_q^X \\
& && \text{operations using operator type } q \text{ are occupied in each} \\
& && \text{congruence class modulo } \Pi^X \quad (\rightarrow 2.2) \\
& && n^r(A^X) \leq N_r \quad \forall r \in R \quad (\rightarrow 5.3) \\
& && a_q^X \in \mathbb{N}_0, \text{ and } a_q^{\min} \leq a_q^X \leq a_q^{\max} \quad \forall q \in Q
\end{aligned}$$

Figure 5.2: Template model for resource-aware modulo scheduling

in the base formulation with integer decision variables modelling the allocation. For notational convenience, we consider these variables to be part of an intermediate solution X . Then, one would minimise the ILP according to the objective function $f^{\text{RU}}(X)$. The specific changes required to extend state-of-the-art schedulers are described in the following. The formulations considered in this chapter expect and leverage the fact that all shared operator types $q \in Q^{\text{Sh}}$ are fully pipelined and thus have $b_q = 1$.

5.2.1 Formulation by Eichenberger and Davidson

The EDFORM limits the use of an operator type per modulo slot only on the right-hand sides of constraints (2.14) in Section 2.8.1. To that end, introducing appropriate allocation variables as in (5.12), and changing the objective are thus the only changes required to their model.

In (5.12), we also simplified the constraints to assume fully-pipelined operator types.

$$\sum_{i \in O^q} m_i^x \bmod \Pi^X \leq a_q^X \quad \forall x \in [0, \Pi^X - 1], \forall q \in Q^{\text{Sh}} \quad (5.12)$$

5.2.2 Formulation by Šůcha and Hanzálek

The SHFORM is the only formulation for which a resource-aware extension was already proposed [86]. We reimplemented their unit-processing time formulation, as presented in Section 2.8.2, to be used in our MORAMS approach. The resource-aware variant simply exchanges constraints (2.28) by (5.13).

$$\sum_{j \in O^q: i \neq j} \mu_{ij}^{(y)} \leq a_q^X - 1 \quad \forall i \in O^q, \forall q \in Q \quad (5.13)$$

5.2.3 Moovac formulation

Both variants of the Moovac formulation, as presented in Section 3.1, require the same changes to be made resource-aware. The allocation parameters occur in constraints (3.4), (3.5) and (3.11). In the latter,

the parameters can be replaced with the the corresponding decision variables (5.14). Doing the same in (3.4) and (3.4) would result in a product of variables, though. However, we can linearise these constraints by replacing the occurrence of a_q , which serves as a big-M constant here, with the maximum allocation, a_q^{\max} , as highlighted in (5.15) and (5.16).

$$w_i \leq a_{\sigma(i)}^X - 1 \quad \forall i \in O^{\text{Sh}} \quad (5.14)$$

$$w_j - w_i - 1 - (\omega_{ij} - 1) \cdot a_q^{\max} \geq 0 \quad \forall (q, i, j) \in \mathcal{D} \quad (5.15)$$

$$w_j - w_i - \omega_{ij} \cdot a_q^{\max} \leq 0 \quad \forall (q, i, j) \in \mathcal{D} \quad (5.16)$$

CONFLICT-COUNTING OPERATOR CONSTRAINTS In its current form, the MOOVAC formulation computes a binding, i.e. mapping of operations to concrete operators, in contrast to EDFORM and SHFORM, which only ensure that no more than the allocated number of operators are used in each modulo congruence class. For a fairer comparison, we adopted Šucha and Hanzálek's idea of counting the modulo slot conflicts among the operations competing for the same shared operator type.

To this end, we drop the variables w_i and ω_{ij} , constraints (3.3) and (3.9), as well as the newly modified constraints (5.14)–(5.16) from the formulation. Instead, we add the constraints (5.17).

This modelling is only suitable for operator types with a blocking time of 1. A more general variant will be presented in Section 6.3.

$$\sum_{j \in O^q, i \neq j} 1 - \mu_{ij} - \mu_{ji} \leq a_q^X - 1 \quad \forall i \in O^q, \forall q \in Q^{\text{Sh}} \quad (5.17)$$

Recall that the binary variables μ_{ij} and μ_{ji} are both zero if and only if operations i and j are assigned to the same congruence class, and thus are in conflict. For example, let an operation be in conflict with h other operations. Then we need at least $h + 1$ operators for a feasible solution. Note that there are $|O^q|$ -many instances of (5.17) that work together similar to a maximum constraint: the operation with the highest number of conflicts determines the required allocation for the operator type.

5.3 APPROACHES FOR THE MORAMS PROBLEM

In the following, we discuss two different approaches to solve the MORAMS problem, i.e. computing a set \mathcal{S} of Pareto-optimal solutions regarding $f^{\text{II}}(X)$ and $f^{\text{RU}}(X)$, with the help of the RAMS formulations described above.

5.3.1 ε -Approach

The ε -approach is a standard method from the multi-criteria optimisation field [23]. Its core idea, given two objectives, is to optimise for only

Algorithm 2 ε -approach for the MORAMS problem

```

1: Let ILP be an exact modulo scheduling formulation with decision
   variables  $\Pi^X, a_q^X, \forall q \in Q$ , and  $t_i^X, \forall i \in O$ 
2: ILP.construct( )
3:  $\mathcal{S} = \emptyset$ 
4: loop
5:    $S = \text{new solution}$ 
6:   ILP.solveWithObjective(  $f^\Pi(X)$  )
7:   if solver status is not optimal then
8:     stop exploration
9:    $\Pi^S = \text{ILP.value}( \Pi^X )$ 
10:  ILP.addConstraint(  $\Pi^X == \Pi^S$  )
11:  ILP.solveWithObjective(  $f^{\text{RU}}(X)$  )
12:  if solver status is not optimal then
13:    stop exploration
14:   $a_q^S = \text{ILP.value}( a_q^X ) \forall q \in Q$ 
15:   $t_i^S = \text{ILP.value}( t_i^X ) \forall i \in O$ 
16:  ILP.removeConstraint(  $\Pi^X == \Pi^S$  )
17:  ILP.addConstraint(  $f^\Pi(X) \geq f^\Pi(S) + 1$  )
18:  ILP.addConstraint(  $f^{\text{RU}}(X) \leq f^{\text{RU}}(S) - \min_{r \in R} \frac{1}{N_r \cdot |R|}$  )
19:   $\mathcal{S} = \mathcal{S} \cup \{S\}$ 
20: return  $\mathcal{S}$ 

```

one objective, and successively add constraints for the other. In order to apply the method for solving the MORAMS problem, we need to employ a RAMS formulation where *all* components of a solution are decision variables, such as the Moovac-I formulation with the extensions discussed above. Algorithm 2 shows pseudocode outlining our implementation.

STANDARD METHOD The basic functionality of the ε -approach is realised by the following steps. First, an extreme point according to $f^\Pi(X)$ is computed (Line 6), and a solution S is extracted (Lines 9, 14, 15). Then, for the next iteration, a constraint forcing the resource utilisation to be less than its current value *minus an ε* , is added (Line 18), and the model is again solved with the Π minimisation objective (Line 6). We use $\varepsilon = \min_{r \in R} \frac{1}{N_r \cdot |R|}$, i.e. the smallest possible decrease in the objective value according to the device resources. This algorithm is iterated until the successively stronger ε -constraints prevent any new feasible solution to be discovered.

EXTENSION FOR FASTER CONVERGENCE We deviate slightly from the standard method by lexicographically minimising both the Π and the resource utilisation, to ensure that we obtain the smallest possible allocation for each interval. This is achieved by fixing Π^X to

its value after the Π -minimisation (Line 10), and then optimising for $f^{\text{RU}}(X)$ (Line 11). Afterwards, the constraint on the Π is removed again (Line 16).

As a bonus, we know that the Π will increase in each iteration, and encode this insight in the form of a second, non-standard ε -constraint regarding $f^{\Pi}(X)$ (Line 17). Here, the ε -value is 1, as the Π^X is an integer variable.

SOLUTIONS We only accept ILP solutions that were proven to be optimal by the solver, as suboptimal solutions could yield dominated MORAMS solutions and interfere with the convergence of the algorithm. Conversely, the set of solutions \mathcal{S} returned in Line 20 is guaranteed to only contain Pareto-optimal solutions, and no post-filtering is needed.

5.3.2 Iterative Approach

This approach is applicable to the far larger class of modulo schedulers that try different candidate Π s until the first feasible solution is found.

As an alternative to the ε -approach that requires the Π to be a decision variable, we propose an iterative approach, in which the Π is a constant parameter for each iteration, to tackle the MORAMS problem.

This approach is outlined in Algorithm 3. We choose successively larger candidate Π s from the range of possible intervals (Line 4), construct the ILP parameterised to that Π (Line 7), solve it with the resource utilisation objective (Line 8) and, given that the ILP solver has proven optimality, retrieve and record the solution (Lines 15 to 17). In case the solver proves the infeasibility of an attempt, we continue with the next candidate Π (Line 11), as in this situation, not enough operators can be allocated within the given resource constraints to make the current candidate Π feasible. We stop the exploration if the solver returns either no solution, or a suboptimal one, due to a violated time limit.

Dominated solutions are filtered in place (Line 18). Due to the strictly monotonically increasing progression of the candidate Π s, it is sufficient to compare the current solution to the last Pareto-optimal solution S^{last} (if available). Compared to S^{last} , the current solution S needs to have a lower resource utilisation in order to be added to the result set.

While filtering out the dominated solutions is easy in our setting, significant time may be wasted in computing them. To this end, we propose two heuristic rules to skip scheduling attempts that would result in obviously dominated solutions.

CANDIDATE-SKIPPING RULE The first rule is shown in Lines 5 to 6. As discussed above, the iterative exploration with increasing candidate Π s means that we can detect dominated solutions by inspecting their resource utilisation. Now, if we know *a priori* that the resource

Algorithm 3 Iterative approach to the MORAMS problem

```

1: Let ILP be an exact modulo scheduling formulation with a candidate interval  $\Pi^X$  (a parameter), and decision variables  $\alpha_q^X \forall q \in Q$  and  $t_i^X \forall i \in O$ 
2:  $\mathcal{S} \leftarrow \emptyset$ 
3:  $S^{\text{last}} \leftarrow \text{null}$ 
4: for  $\Pi^X \in [\Pi^{\min}, \Pi^{\max}]$  do ▷ Iterate in ascending order
5:   if  $S^{\text{last}} \neq \text{null}$  and  $A^{S^{\text{last}}} = A^{\text{tr}}(\Pi^X)$  then
6:     continue with next candidate  $\Pi$ 
7:   ILP.construct(  $\Pi^X$  )
8:   ILP.solveWithObjective(  $f^{\text{RU}}(X)$  )
9:   if solver status is infeasible then
10:     $S^{\text{last}} \leftarrow \text{null}$ 
11:    continue with next candidate  $\Pi$ 
12:   else if solver status is not optimal then
13:     stop exploration
14:    $S \leftarrow \text{new solution}$ 
15:    $\Pi^S \leftarrow \Pi^X$ 
16:    $\alpha_q^S \leftarrow \text{ILP.value}( \alpha_q^X ) \forall q \in Q$ 
17:    $t_i^S \leftarrow \text{ILP.value}( t_i^X ) \forall i \in O$ 
18:   if  $S^{\text{last}} = \text{null}$  or  $f^{\text{RU}}(S) < f^{\text{RU}}(S^{\text{last}})$  then
19:      $\mathcal{S} \leftarrow \mathcal{S} \cup \{S\}$ 
20:      $S^{\text{last}} \leftarrow S$ 
21:   if  $A^S = A^{\min}$  then
22:     stop exploration
23: return  $\mathcal{S}$ 

```

utilisation for a given candidate Π cannot be improved over the last Pareto-optimal solution, we can skip such a candidate Π completely. To that end, we check whether the last solution's allocation, $A^{S^{\text{last}}}$, is equal to the trivial allocation for the current candidate interval, $A^{\text{tr}}(\Pi^X)$. If so, we skip the candidate Π without invoking the ILP solver.

Li et al. used a similar rule to filter candidate Π s' based on the respective trivial allocations [54]. However, their definition disregards the possibility that these allocations may be infeasible, and therefore can lead to incorrectly excluded candidate Π s.

Recall that the trivial allocation represents the theoretical minimum number of operators required to make an Π feasible.

EARLY-TERMINATION RULE The second rule (Lines 21 to 22) stops the exploration if the minimum allocation A^{\min} is achieved. All remaining solutions would be dominated by the current solution because the allocation cannot be improved further, and those solutions would have larger Π s.

Note that both rules can only be applied if the respective minimal

Table 5.3: Problem sizes

Number of ...	min.	median	mean	max.
operations	14	49	104	1374
shared operations	0	4	16	416
edges	17	81	237	4441
backedges	0	3	23	1155

allocations are feasible, which may not be the case in the presence of deadlines imposed by either backedges or latency constraints.

5.3.3 Dynamic Lower Bound for the Allocation

In order to make it easier for the ILP solver to prove that it has reached the optimal allocation for the current Π , we propose to include the bound (5.10) in the models. When using the iterative approach, we can simply add it as a linear constraint to the formulation, since Π^X is a constant. For the ε -approach, however, (5.10) would be a quadratic constraint. To linearise it, we introduce binary variables π^x that represent a particular value of Π^X according to (5.18)–(5.19).

$$\sum_{x \in [\Pi^{\min}, \Pi^{\max}]} x \cdot \pi^x = \Pi^X \quad \sum_{x \in [\Pi^{\min}, \Pi^{\max}]} \pi^x = 1 \quad (5.18)$$

$$\pi^x \in \{0, 1\} \quad \forall x \in [\Pi^{\min}, \Pi^{\max}] \quad (5.19)$$

With the help of these variables, we can now impose constraints (5.20), inspired by inequality (2.11).

$$x \cdot a_q^X \geq \pi^x \cdot |O^q| \cdot b_q \quad \forall x \in [\Pi^{\min}, \Pi^{\max}], \forall q \in Q^{\text{Sh}} \quad (5.20)$$

5.4 EVALUATION

We evaluated the presented MORAMS approaches on a set of 204 realistic test instances. These modulo scheduling problems were extracted from two different HLS environments: 16 instances originate from Simulink models compiled by the Origami HLS project [70], whereas 188 instances represent loops from the well-known C-based HLS benchmark suites CHStone [38] and MachSuite [75]. The latter were compiled by the Nymbler C-to-hardware compiler as described in [66], using an operator library from the Bambu HLS framework [71]. Table 5.3 summarises the problem sizes. Our target device was the Xilinx Zynq XC7Z020, a popular low-cost FPGA found on several evaluation boards. As resources, we model the FPGA's number of lookup tables (53200), DSP slices (220), and, specifically for the C-based benchmark instances, assume the availability of up to 16 memory ports that

can be used to either read from or write to an address space shared with the ARM CPU-based host system of the Zynq device.

We performed the proposed design-space exploration using Gurobi 8.1 as ILP solver on 2×12 -core Intel Xeon E5-2680 v3 systems running at 2.8 GHz with 64 GiB RAM. The schedulers were allowed to use up to 8 threads, 6 hours wall-clock time and 16 GiB of memory *per instance*. We report each instance’s *best* result from two runs, considering first the number of solutions, and then the accumulated runtime of the exploration.

As discussed in Section 5.1.2, we consider the latency T^{\max} as a separate user constraint here. We scheduled our test instances subject to three different latency constraints that cover the whole spectrum of cases: The strongest constraint is to limit the schedule length T^{\max} to the length of the critical path T^{CP} . Using the length of a non-modulo schedule with heuristic resource constraints, T^{NM} , relaxes the operations’ deadlines slightly. Lastly, we adapt the loose but conservative bound T^{IM} from Section 3.2.2 to consider the minimum allocation, which by construction does not exclude any modulo schedule with minimal length.

In the following discussion, \mathcal{S} denotes the set of Pareto-optimal solutions for a given instance of the MORAMS problem. Let the set \mathcal{S}^{co} contain all solutions computed by a particular approach, including the ones that were immediately discarded as dominated in the iterative approach. Additionally, we define the set $\mathcal{S}^{\text{tr}} \subseteq \mathcal{S}$ of trivial solutions, by which we mean solutions S that have the trivial allocation for their respective Π , formally $A^S = A^{\text{tr}}(\Pi^S)$.

Figure 5.3 illustrates these metrics and the shape of the solution space resulting from the exploration with our iterative approach for the instance representing the Simulink model `splin_pf`. We picked this particular instance because it behaves differently under the three latency constraints, and showcases the effects of our heuristic rules. In the case $T^{\max} = T^{\text{CP}}$, many dominated solutions were computed because the minimal allocation A^{\min} was not feasible, and consequently, the early-termination rule was not applicable. Also, the candidate-skipping rule was only able to skip candidate Π s 6–7. For $T^{\max} = T^{\text{NM}}$, the situation was significantly relaxed, as we only computed one dominated solution at $\Pi = 8$, and were able stop the exploration at $\Pi = 9$. Lastly, with $T^{\max} = T^{\text{IM}}$, all solutions were trivial, and no extra dominated solutions were computed. The equivalent plots for the ϵ -approach, which we omit here for brevity, only contain the orange-coloured Pareto-optimal solutions by construction. All approaches completed the exploration for `splin_pf` within three seconds of runtime.

The results of the exploration across all 204 test instances are summarised in Table 5.4 for the ϵ -approach of Section 5.3.1, as well as the iterative approach of Section 5.3.2 together with the EDFORM, SHFORM or Moovac-S formulations. The scheduler runtimes are accumulated

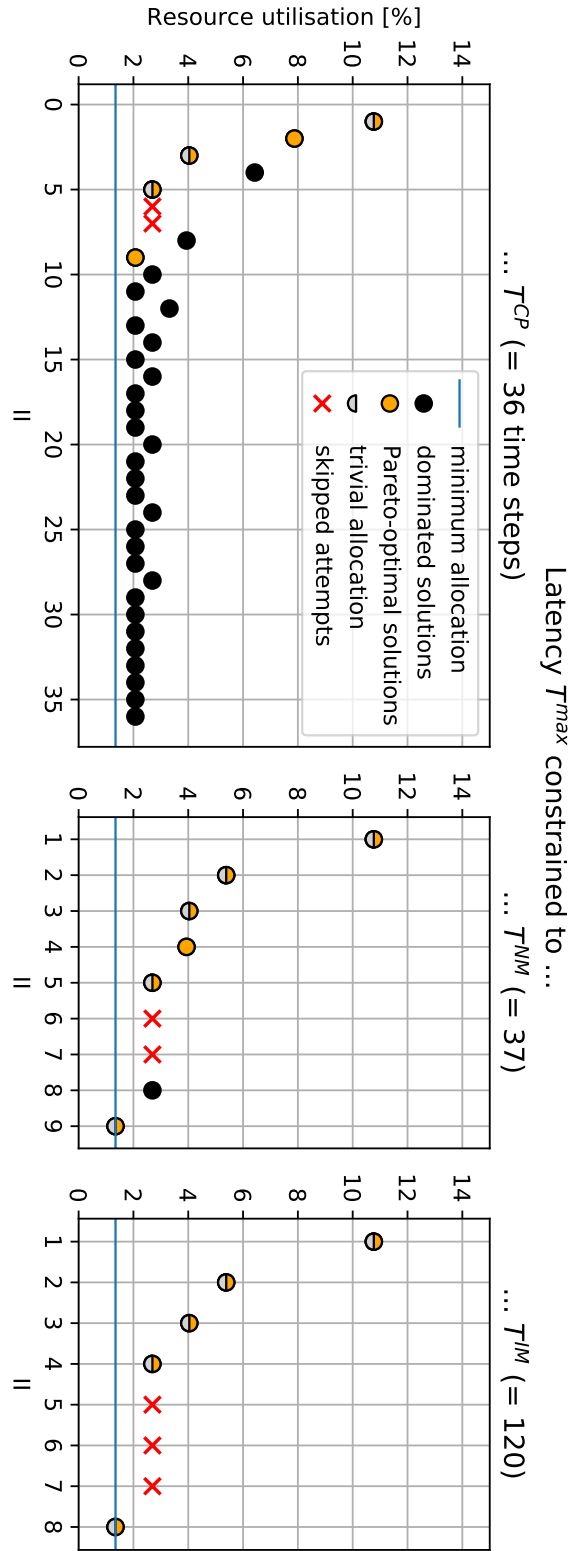


Figure 5.3: Trade-off points for instance `splin_pf`, computed with the iterative approach

Table 5.4: Design-space exploration results for 204 instances

Method	$T^{\max} = T^{\text{CP}}$				$T^{\max} = T^{\text{NM}}$				$T^{\max} = T^{\text{IM}}$			
	RT [h]	$ \mathcal{S}^{\text{co}} $	$ \mathcal{S} $	$ \mathcal{S}^{\text{tr}} $	RT [h]	$ \mathcal{S}^{\text{co}} $	$ \mathcal{S} $	$ \mathcal{S}^{\text{tr}} $	RT [h]	$ \mathcal{S}^{\text{co}} $	$ \mathcal{S} $	$ \mathcal{S}^{\text{tr}} $
ε -approach	12.2	285	285	168	48.4	372	372	302	70.6	321	321	290
<i>iterative:</i>												
EDFORM	2.4	1510	290	170	26.4	498	453	381	34.9	441	422	382
SHFORM	16.2	1502	289	170	48.1	448	412	341	47.7	416	408	371
MOOVAC-S	16.0	1492	289	170	48.2	422	379	308	54.3	353	346	312
RT [h] = “accumulated runtime in hours”.												

in the columns “RT [h]” to give intuition into the computational effort required by the different approaches. Note that in practice, one would not need to schedule a set of instances sequentially. We then count the number of solutions in the aforementioned categories.

According to the complete exploration, the clear winner is the resource-aware EDFORM within our problem-specific, iterative approach, as it computes the most Pareto-optimal solutions (columns “ $|\mathcal{S}|$ ”) in the shortest amount of time (columns “RT [h]”), across all latency constraints, by a large margin. The SHFORM performs slightly better than the MOOVAC-S formulation in the MORAMS setting. We observe that for the tightest latency constraint T^{CP} , fewer trivial allocations are feasible than for the other bounds, which causes the iterative approaches to compute $|\mathcal{S}^{\text{co}}| \gg |\mathcal{S}|$, due to the non-applicability of the heuristic tweaks in Algorithm 3. On the other hand, the fact that $|\mathcal{S}| > |\mathcal{S}^{\text{tr}}|$ demonstrates that only considering solutions with the trivial allocation for the respective Π (e.g. as suggested in [29]) would, in general, not be sufficient to perform a complete exploration.

By design, the ε -approach computes only the Pareto-optimal solutions, regardless of the latency constraint (columns “ $|\mathcal{S}^{\text{co}}|$ ” \equiv “ $|\mathcal{S}|$ ”). However, this benefit is apparently outweighed by the additional complexity introduced by modelling the Π as a decision variable in the MOOVAC-I formulation, causing the ε -approach to be outperformed by the EDFORM.

Note that the accumulated runtimes increase from T^{CP} to T^{IM} across all methods. While on the one hand, a tight bound such as T^{CP} makes the operator-constrained scheduling part of the problem harder, on the other hand it also restricts the ILP search space and thus helps the ILP solver to prove the optimality of a solution faster. As we accept only provably optimal solutions during the exploration, it is apparent that the effects of the second aspect play a greater role in our experiment.

5.5 CHAPTER SUMMARY

We presented a framework to perform a scheduler-driven design-space exploration in the context of high-level synthesis. Despite of leveraging ILP-based modulo scheduling formulations, the MORAMS problem can be tackled in a reasonable amount of time, and yields a variety of throughput vs. resource utilisation trade-off points.

SKYCASTLE: A RESOURCE-AWARE MULTI-LOOP SCHEDULER

Modern **FPGAs** have become large enough to accommodate far more functionality than one simple computational kernel, opening up new opportunities and challenges for designers. For example, when using all available resources, complex multi-phase kernels can be implemented within a single accelerator to reduce the number of context switches [44, 82]. On the other hand, it is also reasonable to partition the resources, e.g. to replicate an accelerator for parallel processing [50], or to share one device among different groups in a research project [92]. In all of the aforementioned situations, the question is usually the same:

How to maximise the performance within the given resource constraints?

In this chapter, we present **SKYCASTLE**, a resource-aware multi-loop scheduler that can answer the question above automatically, and outline how it enables a new way to design hardware accelerators.

This chapter is based on:

[68] Julian Oppermann, Lukas Sommer, Lukas Weber, Melanie Reuter-Oppermann, Andreas Koch and Oliver Sinnen. ‘SkyCastle: A Resource-Aware Multi-Loop Scheduler for High-Level Synthesis’. In: *International Conference on Field-Programmable Technology, FPT*. 2019

6.1 BACKGROUND

The initially stated question implies an optimisation problem. **HLS** tools are an ideal starting point to tackle it, as they can construct microarchitectures with different trade-offs for the accelerator’s performance and resource demand from the *same* algorithmic specification. This work targets HLS tools that accept C/C++ code as input. We argue that the most influential control knob in this context is the amount of *pipelining* (cf. Section 2.2.3) used in the microarchitecture. Recall that a smaller **II** results in more overlapping of iterations and in consequence, in a shorter execution time for the whole loop, but also requires more resources as less operator sharing is possible.

Pipelining is also applicable to functions, where it results in an overlapping evaluation of the function’s body for different sets of arguments. The same trade-off considerations and scheduling techniques apply to both forms of pipelining, though.

Listing 6.1: Sum-Product Network example

```

double spn(...)          { /* 10 FP mul, 1 FP add */ }
double spn_marginal(...) { /* 8 FP mul, 1 FP add */ }

double top(char i1, char i2, char i3, char i4) {
    // most probable explanation for "i5"
    char maxClause = -1; double maxProb = -1.0;
    MPE: for (char x = 0; x < 0xFF; x += 4) {
        double p0 = spn(i1, i2, i3, i4, x);
        double p1 = spn(i1, i2, i3, i4, x+1);
        double p2 = spn(i1, i2, i3, i4, x+2);
        double p3 = spn(i1, i2, i3, i4, x+3);
        maxProb = ... // max(maxProb, p0, p1, p2, p3);
        maxClause = ... // argument value for i5 that
                        // yielded new value for maxProb
    }
    double pM = spn_marginal(i2, i3, i4, maxClause);
    return maxProb / pM;
}

```

6.1.1 Motivational Example

Consider the excerpt from the inference process in a Sum-Product Network (SPN) (see also Section 6.5.1) in Listing 6.1. We instruct Xilinx Vivado HLS to pipeline the loop labeled MPE, which automatically pipelines the function `spn` as well. The function `spn_marginal` will be inlined automatically by the HLS frontend. Vivado HLS attempts, and succeeds, to construct the maximum performance version of this kernel with $\Pi=1$ for the loop and the function. However, as this results in a fully-spatial microarchitecture, each operation in the computation requires its own operator. When targeting the popular ZedBoard, such a design requires 499 DSP slices, which exceeds the available 220 slices by a large margin. Finding the lowest-latency version that still fits on the device requires considering a) the degree of pipelining applied to function `spn`, b) the number of `spn`-instances, c) the amount of pipelining for loop MPE (which depends on a) and b)), and lastly, d) the operator allocation for the top-level function, which influences c) as well as the latency of the non-pipelined computation at the end of `top`. Here, the fastest solution is to pipeline `spn` and MPE with $\Pi=4$, allocate two multipliers, one adder, one divider, three floating-point comparators and four instances of `spn` inside the function `top`.

6.1.2 Approach and Contributions

This chapter makes the following key contributions.

First, we provide the formal definition of the *Multi-Loop Scheduling (MLS)*: an integrated scheduling and allocation problem that models the interaction between these two core steps (cf. Section 2.1.3) for HLS kernels containing arbitrarily nested loops and functions. The MLS problem serves as the theoretical foundation for solving the aforementioned optimisation problem.

Secondly, we present SKYCASTLE, a resource-aware multi-loop scheduler capable of solving the problem for a subclass of kernels composed of multiple, nested loops in a single top-level function.

Both the proposed problem definition and the scheduler apply to, or can be easily adapted to, any HLS flow. However, in order to demonstrate the practical applicability of the approach, we tailored the scheduler to be *plug-in* compatible with the Vivado HLS engine. To that end, we faithfully extract the actual scheduling and allocation problems faced by Vivado HLS from its intermediate representation. Afterwards, we feed the directives required to control pipelining and the operator allocation according to the solutions determined by our scheduler back to the synthesis flow.

Vivado HLS' default settings aim at maximum performance but may fail in later synthesis steps due to resource demands that exceed the capacity on the target device. Even with our proof-of-concept implementation, we are able to guide Vivado HLS to generate synthesisable microarchitectures for three complex kernels on two FPGA devices. On the larger device, we also explore partitioning the available resources in order to enable the replication of slightly slower, but smaller accelerators as a means to further boost the overall performance. The multi-accelerator solution easily outperforms the *theoretical* maximum-performance, single-accelerator design, which is actually unsynthesisable for two of our three case studies.

Vivado HLS' proprietary nature currently prevents a deeper integration into the synthesis flow.

6.1.3 Related Work

We discuss the related work with regards to the kind of exploration used to discover solution candidates to answer the initially stated research question. For a general overview of modulo scheduling approaches, refer to Section 2.7.

6.1.4 As Part of the HLS Scheduler

The most direct way to solve the problem is to model it inside the HLS scheduler. This requires considering the highly interdependent problems of scheduling and (operator) allocation together, but has two main benefits: First, the resulting schedules are guaranteed to be feasible because they were computed by an actual scheduler that considers all nuances of the problem, such as tight inter-iteration

dependences that might require more operators than the theoretical lower bound. Secondly, no external exploration is required.

Our **MORAMS** study from Chapter 5 is the only work that fits in this category, but it is not sufficient to answer the problem directly, because we only modulo-scheduled *individual* loops under the assumption of an independent operator allocation, instead of more complex multi-loop kernels. However, our proposed scheduler builds upon the **RAMS** framework and can be seen as a significant extension of the previous chapter, in order to suit a more practical context.

6.1.5 *Pipelining-focussed Exploration*

The next category is comprised of approaches that control the amount of pipelining in a complex kernel by determining target IIs for its pipelined parts, e.g. stages in a pipelined streaming application [51], stateless actors in a synchronous data-flow graph [12, 13], or loops arranged in a directed, acyclic graph [54]. Common aspects in these works are a) the use of a performance model to choose the IIs, b) the approximation of latencies of the individual parts, and c) the derivation of the operator allocation from the II, without checking the feasibility.

Differences exist in the chosen objectives. Li et al. [54] tackle a problem very similar to ours: minimise the overall latency of a kernel, subject to low-level resource constraints, and consider the benefits of slightly slower, but better replicable implementations.

Cong et al. [12] and Cong, Huang and Zhang [13] and Kudlur, Fan and Mahlke [51] attempt to minimise the required resources to fulfil an externally given throughput constraint, and, in consequence, would need some kind of exploration to find the highest throughput that still satisfies given resource constraints. Note, though, that these approaches employ more elaborate models of generated microarchitectures than we do. For example, the cost-sensitive modulo scheduler [29] used in [51] considers the different bitwidths of operations as well as the required interconnects and register storage, but crucially, performs the *allocation* of functional units *before* scheduling.

6.1.6 *General Design-Space Exploration*

General design-space exploration approaches form the last (and largest) category, whose representatives may be model-based analysis tools [96, 98], integrated in an HLS flow [30, 73], or consider the HLS tool as a black box and emit directives to control the microarchitecture generation [77]. These approaches usually consider other techniques besides pipelining, such as loop unrolling, function inlining, or partitioning of arrays. Most tools aim to explore a diverse set of solutions for the (human) designer choose from. A notable exception is the

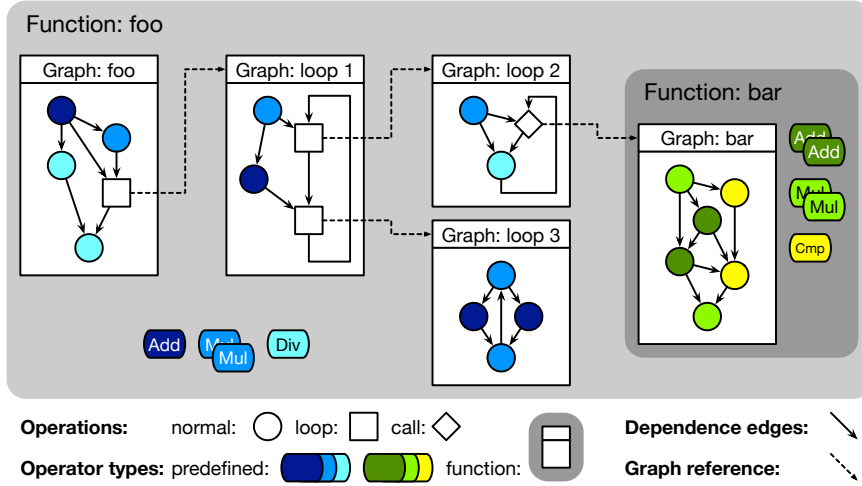


Figure 6.1: An example instance of the multi-loop scheduling problem

work of Prost-Boucle, Muller and Rousseau [73], which describes an autonomous flow that successively applies transformations to improve the kernel’s latency while obeying low-level resource constraints. However, internally, the allocation of operators precedes the scheduling phase.

6.2 THE MULTI-LOOP SCHEDULING PROBLEM

6.2.1 Informal Problem Description

Given an HLS kernel in a structured programming language, composed of multiple, optionally pipelined, loops and functions, we want to minimise the latency of one activation of the kernel’s unique top-level function, subject to resource constraints in terms of the low-level FPGA resources, e.g. look-up tables or DSP slices.

Figure 6.1 shows an example instance of the MLS problem. We have several dependence graphs that each correspond to the body of a loop in the kernel, derived e.g. from a CDFG representation inside the HLS tool. The non-loop parts of functions are treated uniformly as single-iteration loops at the outermost level. In addition to what is shown in the example, the general problem definition in Section 6.2.3 will allow multiple, different functions to be called from any graphs of a function. Also, any function may contain arbitrary loop structures.

Our goal in the *scheduling* part of the problem is to compute start times for each operation, and to determine a feasible initiation interval for graphs originating from pipelined parts of the kernel.

The operations in the graphs require operators, which occupy a specific amount of the FPGA’s resources. HLS tools may *share* operators

among several operations if the resource demand of the operator is higher than the cost of the additional multiplexing logic. Determining the number of operators of each type constitutes the *allocation* part of the problem, and has a strong influence on the scheduling result.

We introduce the concept of an *allocation domain*, which provides the operators for a subset of the graphs. All graphs in an allocation domain share these operators, but assume exclusive access to them. This means that the parts of the computation represented by any pair of graphs in the same allocation domain will be executed sequentially at runtime. In contrast, graphs in different allocation domains can execute in parallel due to their independent sets of operators.

Figure 6.1 also presents the canonical examples for these concepts, inspired by Vivado HLS, which implements operator sharing at the function level. Here, the two functions, `foo` and `bar`, represent the allocation domains. The former contains four graphs, i.e. the function body plus the three loops `loop1`–`loop3`. The operations in these graphs share the allocated operators within `foo`. Function `bar` contains only one graph, `bar's` body. Nested loops are represented by special operations (squares in the figure) that reference another graph in the same allocation domain. Lastly, the function call (rhombus in `loop2`) references another graph embedded in its own allocation domain (= `bar`), which needs to be instantiated as a special operator type in the surrounding allocation domain (= `foo`).

We will implicitly assume these correspondences for the rest of this chapter, and name the special operations and operators accordingly in order to keep the following problem definition as intuitive as possible. Note, however, that the underlying modelling ideas apply to other resource/operator sharing strategies as well, e.g. sharing only within the same loop level, in which case each graph would be embedded into its own allocation domain.

6.2.2 Extended Notation

Tables 6.1 to 6.3 summarise the notation used in this chapter, which extends the formalism from the previous chapters in a natural way. Up until now, instances of the scheduling and allocation problems at hand represented individual loops, and were comprised of operations and edges in a single dependence graph, and a set of operator types partitioned into shared and unlimited types. In order to define the problem signature for an instance of the MLS problem in Section 6.2.3, we additionally need to distinguish

- multiple functions with their individual subsets of graphs and operator types,
- predefined and function operator types,
- pipelined and non-pipelined graphs, and

- normal, loop and call operations.

The set \mathcal{F} contains all functions of the kernel, and partitions the sets of dependence graphs G and operator types Q : a function $F \in \mathcal{F}$ is associated with its unique subsets of graphs G_F and operator types Q_F . The reverse mapping from a given graph or operator to the surrounding function is established by the function φ . We define $\varphi(g) = F \Leftrightarrow g \in G_F$, and analogously, $\varphi(q) = F \Leftrightarrow q \in Q_F$.

We assign a latency l_q to each operator type $q \in Q$. *Predefined* operator types (Q^{Pd}) have static characteristics extracted from the HLS tool's operator library. In contrast, a *function* operator type $q \in Q^{Fu}$ references a graph $\gamma(q)$ in another function, and derives its latency, blocking time, and resource demands from the other graph's intermediate scheduling and allocation result. We continue to distinguish *shared* (Q^{Sh}) and *unlimited* (Q^∞) operator types. Note that function operator types are always considered to be shared. For notational convenience, we define subsets $Q_F^{Sh}, Q_F^\infty, Q_F^{Pd}, Q_F^{Fu}$ of these classification sets per function F . The symbol a_q still denotes the allocated number of instances of operator type $q \in Q$. However, as we now need to distinguish one allocation *per function*, we let A_F group together the individual allocations for the operator types in a function F .

We assume to have a constant known trip count c_g for each graph $g \in G$, and let T_g denote its latency. The set G^{Pl} marks graphs that correspond to a pipelined loop or function in the kernel. A pipelined graph $g \in G^{Pl}$ has an initiation interval Π_g . We denote the kernel's top-level graph as g^{top} .

Each graph $g \in G$ is defined by its set of operations O_g , and set of dependence edges E_g . We distinguish three kinds of operations. A *normal* operation $i \in O_g^{No}$ is associated with a predefined operator type $\sigma(i)$. A *loop* operation $i \in O_g^{Lo}$ represents a nested loop in the graph, and references another graph $\gamma(i)$ in the same function. Lastly, a *call* operation $i \in O_g^{Ca}$ models a function call in the kernel. It is associated with a function operator type $\sigma(i)$, which in turn references the graph $\gamma(\sigma(i))$ corresponding to the body of the called function.

The definitions above use the function σ to map an operation to its operator type, and γ to map a loop operation or function operator type to the referenced graph.

The sets O_g^q represent a graph g 's subset of operations that use a particular operator type q , according to $\sigma(i) = q \Leftrightarrow i \in O_g^q$ for every normal or call operation $i \in O_g^{No} \cup O_g^{Ca}$.

The remaining symbols in Tables 6.1 to 6.3 carry the same meaning as before.

Here, the latency is synonymous to the schedule length.

Normal operations are equivalent to the only kind of operations in previous chapters.

6.2.3 Formal Problem Definition

Figure 6.2 defines the signature of the MLS problem. The input is comprised of the specification of the model components according to

Table 6.1: Notations to describe the MLS problem: Sets

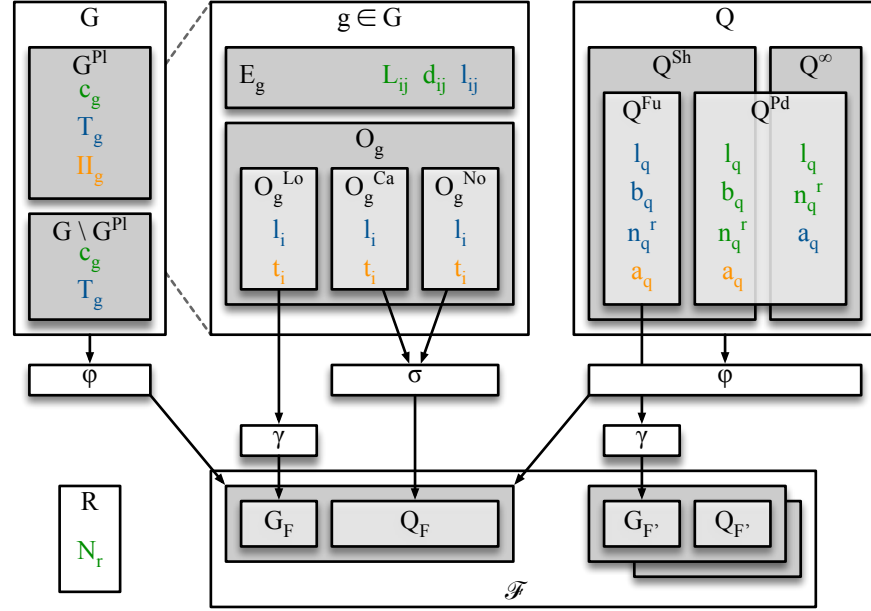
Symbol	Description
R	Set of resource types
Q $= Q^{\text{Sh}} \cup Q^{\infty}$ $= Q^{\text{Pd}} \cup Q^{\text{Fu}}$	Set of operator types
Q^{Sh}	Set of shared operator types
Q^{∞}	Set of unlimited operator types
Q^{Pd}	Set of predefined operator types
Q^{Fu}	Set of function operator types
G	Set of graphs
$G^{\text{pl}} \subseteq G$	Set of pipelined graphs
$g^{\text{top}} \in G$	Top-level graph of the kernel
$O_g = O_g^{\text{No}} \cup O_g^{\text{Lo}} \cup O_g^{\text{Ca}},$ $\forall g \in G$	Set of operations in graph g
$O_g^{\text{No}} \forall g \in G$	Set of normal operations in graph g
$O_g^{\text{Lo}} \forall g \in G$	Set of loop operations in graph g
$O_g^{\text{Ca}} \forall g \in G$	Set of call operations in graph g
$O_g^q \subseteq O_g, \forall q \in Q$	Set of operations in g using operator type q
$E_g \forall g \in G$	Set of dependence edges in graph g
\mathcal{F}	Set of functions
$G_F \subseteq G, \forall F \in \mathcal{F}$	Graphs in function F
$G_F^{\text{pl}} \subseteq G, \forall F \in \mathcal{F}$	Pipelined graphs in function F
$Q_F \subseteq Q, \forall F \in \mathcal{F}$	Operator types in function F
$Q_F^{\text{Sh}}, Q_F^{\infty}, Q_F^{\text{Pd}}, Q_F^{\text{Fu}}$	Operator types of a certain kind in function F

Table 6.2: Notations to describe the MLS problem: Attributes

Symbol		Description
N_r	$\in \mathbb{N}_0, \forall r \in R$	Resource limit for type r
l_q	$\in \mathbb{N}_0, \forall q \in Q$	Latency (in time steps) of operator type q
b_q	$\in \mathbb{N}, \forall q \in Q^{\text{Sh}}$	Blocking time (in time steps) of shared operator type q
n_q^r	$\in \mathbb{N}_0, \forall q \in Q, \forall r \in R$	Resource demand of one instance of operator type q , regarding resource type r
a_q	$\in \mathbb{N}_0, \forall q \in Q$	Allocation, i.e. a number of instantiated operators for each type q
A_F	$= \langle a_{q_1}, \dots, a_{q_{ Q_F }} \rangle$	Alternative notation for an allocation regarding function F , grouping together the individual allocations for all of its operator types
$n^r(A)$	$\in \mathbb{N}_0, \forall r \in R$	Accumulated resource demand of an allocation A
c_g	$\in \mathbb{N}, \forall g \in G$	Number of iterations for one invocation of graph g
T_g	$\in \mathbb{N}_0, \forall g \in G$	Latency (in time steps) for graph g
Π_g	$\in \mathbb{N}, \forall g \in G^{\text{Pl}}$	Initiation interval (in time steps) for pipelined graph g
$\forall g \in G :$		
l_i	$\in \mathbb{N}_0, \forall i \in O_g$	Latency (in time steps) of operation i
t_i	$\in \mathbb{N}_0, \forall i \in O_g$	Start time (in time steps) of operation i
d_{ij}	$\in \mathbb{N}_0, \forall (i \rightarrow j) \in E_g$	Edge distance (in number of iterations)
L_{ij}	$\in \mathbb{N}_0, \forall (i \rightarrow j) \in E_g$	Additional edge latency (in number of time steps)
l_{ij}	$\in \mathbb{N}_0, \forall (i \rightarrow j) \in E_g$	Required number of time steps between operations i and j

Table 6.3: Notations to describe the MLS problem: Mappings

Symbol	Description
$\varphi : G \rightarrow \mathcal{F}$	Maps graphs to their surrounding function
$\varphi : Q \rightarrow \mathcal{F}$	Maps operator types to their surrounding function
$\sigma : O_g^{\text{No}} \cup O_g^{\text{Ca}} \rightarrow Q_{\varphi(g)}, \forall g \in G$	Maps normal and call operations to their associated operator type (in the same function)
$\gamma : O_g^{\text{Lo}} \rightarrow G_{\varphi(g)}, \forall g \in G$	Maps loop operations to the graph representing the nested loop (in the same function)
$\gamma : Q_F^{\text{Fu}} \rightarrow G_{F'}, \forall F, F' \in \mathcal{F} : F \neq F'$	Maps function operator types to the graph representing the body of the called function



Attributes are either **parameters**, **derived**, or part of the **solution**.

Figure 6.2: Overview of MLS problem model

the sets in Table 6.1, and the static mappings according to Table 6.3. Additionally, all attributes from Table 6.2 that are in green colour in the Figure 6.2 are input parameters. A solution to the problem consists of a schedule for each graph (6.1), an Π for each pipelined graph (6.2), and the allocation in each function (6.3).

For clarity, we omit the superscript previously used to differentiate solutions here, as we only seek to compute one solution.

$$t_i \quad \forall i \in O_g, \forall g \in G \quad (6.1)$$

$$\Pi_g \quad \forall g \in G^{Pl} \quad (6.2)$$

$$a_q \quad \forall q \in Q_F^{Sh}, \forall F \in \mathcal{F} \quad (6.3)$$

The solution components are highlighted in orange colour in Figure 6.2.

DERIVED ATTRIBUTES The values of the blue attributes in Figure 6.2 are derived from other parameters and solution components. Considering them as part of the problem signature is therefore not mandatory, but simplifies the model.

First, we define an operation's latency in respect to the particular kind of operation. Normal operations adopt the latency of the associated operator type (6.4). Loop operations have a latency equal to the number of time steps required to execute the entire loop once, either with disjoint iterations for non-pipelined loops (6.5), or with overlapping iterations according to the Π for pipelined loops (6.6). Functions are handled as one-iteration loops, therefore only the latency of the referenced graphs needs to be considered (6.7).

$\forall g \in G :$

$$l_i = l_{\sigma(i)} \quad \forall i \in O_g^{No} \quad (6.4)$$

$$l_i = c_{\gamma(i)} \cdot T_{\gamma(i)} \quad \forall i \in O_g^{Lo} : \gamma(i) \notin G^{Pl} \quad (6.5)$$

$$l_i = (c_{\gamma(i)} - 1) \cdot \Pi_{\gamma(i)} + T_{\gamma(i)} \quad \forall i \in O_g^{Lo} : \gamma(i) \in G^{Pl} \quad (6.6)$$

$$l_i = T_{\gamma(i)} \quad \forall i \in O_g^{Ca} \quad (6.7)$$

(6.8) defines our usual shorthand notation for the smallest number of time steps that j can start after i , according to a given edge. A graph's latency is derived from the start times and latencies of its operations (6.9).

$$l_{ij} = l_i + L_{ij} \quad \forall (i \rightarrow j) \in E_g, \forall g \in G \quad (6.8)$$

$$T_g = \max_{i \in O_g} t_i + l_i \quad \forall g \in G \quad (6.9)$$

Function operators have a variable blocking time that is equal either to the latency (non-pipelined functions, (6.10)), or the Π (pipelined functions, (6.11)), of the graph referenced by the operator type. Similarly, they have variable resource demands, which are derived from the accumulated resource demands of the referenced function's allocation

(6.12). Lastly, per definition we allocate as many instances of each unlimited operator types as we have operations using it (6.13).

$$b_q = T_{\gamma(q)} \quad \forall q \in Q^{\text{Fu}} : \gamma(q) \notin G^{\text{Pl}} \quad (6.10)$$

$$b_q = \Pi_{\gamma(q)} \quad \forall q \in Q^{\text{Fu}} : \gamma(q) \in G^{\text{Pl}} \quad (6.11)$$

$$n_q^r = n^r(A_{\varphi(\gamma(q))}) \quad \forall q \in Q^{\text{Fu}} \quad (6.12)$$

$$a_q = \sum_{g \in G_{\varphi(q)}} |O_g^q| \quad \forall q \in Q^{\infty} \quad (6.13)$$

FEASIBILITY CONSTRAINTS A feasible solution must honour all precedence constraints expressed by the dependence edges (6.14), ensure that no shared operator type is oversubscribed at any time (6.15), and obey the given resource constraints for the top-level function (6.16). For pipelined graphs, these constraints resemble the ones from the definition of the RAMS problem in Section 5.1.1. For non-pipelined graphs, backedges can be ignored because they are automatically satisfied, and no modular arithmetic is required to check for an overlap of blocking times, resulting in simplified constraints.

$$\begin{aligned} \forall (i \rightarrow j) \in E_g, \forall g \in G : \\ \begin{cases} t_i + l_{ij} \leq t_j & g \notin G^{\text{Pl}} \\ t_i + l_{ij} \leq t_j - d_{ij} \cdot \Pi_g & g \in G^{\text{Pl}} \end{cases} \end{aligned} \quad (6.14)$$

$$\begin{aligned} \forall q \in Q_F^{\text{Sh}}, \forall g \in G_F, \forall F \in \mathcal{F} : \\ \begin{cases} |\{i \in O_g^q : t_i \leq x < t_i + b_q\}| \\ \leq a_q \quad \forall x \in [0, T_g] & g \notin G^{\text{Pl}} \\ |\{i \in O_g^q : x \in \{(t_i + \beta) \bmod \Pi_g : 0 \leq \beta < b_q\}\}| \\ \leq a_q \quad \forall x \in [0, \Pi_g - 1] & g \in G^{\text{Pl}} \end{cases} \end{aligned} \quad (6.15)$$

$$\forall r \in R : \quad n^r(A_{\varphi(g^{\text{top}})}) \leq N_r \quad (6.16)$$

OBJECTIVE The objective is to minimise the latency of the kernel's top-level function (6.17), which we currently assume to be non-pipelined.

$$\min T_{g^{\text{top}}} \quad (6.17)$$

6.2.4 Bounds

We propose to compute the bounds shown in Table 6.4 before attempting to solve an instance of the MLS problem. Generally, we inherit the definitions from the RAMS framework described in Section 5.1.3, and extend them to recursively handle approximations for loop operations and function operator types.

Table 6.4: Bounds for the MLS problem

Symbols	Description
$\alpha_q^{\min} \leq \alpha_q \leq \alpha_q^{\max} \quad \forall q \in Q_F^{\text{Sh}}, \forall F \in \mathcal{F}$	Minimum and maximum allocation for a shared operator type q in a function F
$l_i^{\min} \leq l_i \leq l_i^{\max} \quad \forall i \in O_g^{\text{Lo}} \cup O_g^{\text{Ca}}, \forall g \in G$	Minimum and maximum latency for a loop or call operation i in a graph g
$T_g^{\min} \leq T_g \leq T_g^{\max} \quad \forall g \in G$	Minimum and maximum latency/schedule length for a graph g
$\Pi_g^{\min} \leq \Pi_g \leq \Pi_g^{\max} \quad \forall g \in G^{\text{Pl}}$	Minimum and maximum Π for a pipelined graph g
$b_q^{\min} \leq b_q \leq b_q^{\max} \quad \forall q \in Q_F^{\text{Fu}}$	Minimum and maximum blocking time for a function operator type q

ALLOCATION The minimum and maximum allocations are computed for each function F . For a shared operator type $q \in Q_F^{\text{Sh}}$, we set $\alpha_q^{\min} = 1$ as in (5.7), and α_q^{\max} to the maximum number of instances that fit within the given resource limits if all other operator types were fixed at their minimum allocation, according to (5.8). During this computation, the variable resource demand (regarding a type $r \in R$) of a function operator type $q \in Q_F^{\text{Fu}}$ is approximated as $n^r(A_{\varphi(\gamma(q))}^{\min})$, i.e. the accumulated r -demand of the referenced function's own minimum allocation.

LATENCY For each graph $g \in G$, the minimum latency T_g^{\min} is defined by the length of the critical path, without considering any operator limits. For loop and call operations, which have a variable latency, we use best-case approximations as shown in (6.18)–(6.20).

$$l_i^{\min} = c_{\gamma(i)} \cdot T_{\gamma(i)}^{\min} \quad \forall i \in O_g^{\text{Lo}} : \gamma(i) \notin G^{\text{Pl}} \quad (6.18)$$

$$l_i^{\min} = (c_{\gamma(i)} - 1) \cdot \Pi_{\gamma(i)}^{\min} + T_{\gamma(i)}^{\min} \quad \forall i \in O_g^{\text{Lo}} : \gamma(i) \in G^{\text{Pl}} \quad (6.19)$$

$$l_i^{\min} = T_{\gamma(i)}^{\min} \quad \forall i \in O_g^{\text{Ca}} \quad (6.20)$$

If g is not pipelined, we set its maximum latency T_g^{\max} to the length of a heuristic non-modulo schedule (cf. Section 2.6.2) at the minimum operator allocation for the surrounding function. Otherwise, any of the previously discussed bounds for the length of a modulo schedule (e.g. $T^{\text{IM}}, T^{\text{NM}}$) can be applied when considering the respective min-

imum allocation. In both cases, we use the corresponding worst-case approximations for the variable latencies, according to (6.21)–(6.23).

$$l_i^{\max} = c_{\gamma(i)} \cdot T_{\gamma(i)}^{\max} \quad \forall i \in O_g^{\text{Lo}} : \gamma(i) \notin G^{\text{Pl}} \quad (6.21)$$

$$l_i^{\max} = (c_{\gamma(i)} - 1) \cdot \Pi_{\gamma(i)}^{\max} + T_{\gamma(i)}^{\max} \quad \forall i \in O_g^{\text{Lo}} : \gamma(i) \in G^{\text{Pl}} \quad (6.22)$$

$$l_i^{\max} = T_{\gamma(i)}^{\max} \quad \forall i \in O_g^{\text{Ca}} \quad (6.23)$$

INITIATION INTERVAL Recall from Section 2.6 that the lower bound for a graph g 's interval Π_g^{\min} is defined as $\max(\Pi_g^{\text{opr}}, \Pi_g^{\text{rec}})$. In order to compute Π_g^{rec} , we use the best-case latency approximations (6.18)–(6.20) as above. For Π_g^{opr} , we plug the surrounding function's maximum allocation $A_{\varphi(g)}^{\max}$, as well as the minimum blocking times b_q^{\min} for function operator types $q \in Q^{\text{Fu}}$, into formula (2.7).

BLOCKING TIME Function operator types $q \in Q^{\text{Fu}}$ have a variable blocking time that corresponds either to the latency or Π of the referenced graph. To that end, we simply propagate the respective lower and upper bounds in (6.24)–(6.27).

$$b_q^{\min} = T_{\gamma(q)}^{\min} \quad \gamma(q) \notin G^{\text{Pl}} \quad (6.24)$$

$$b_q^{\min} = \Pi_{\gamma(q)}^{\min} \quad \gamma(q) \in G^{\text{Pl}} \quad (6.25)$$

$$b_q^{\max} = T_{\gamma(q)}^{\max} \quad \gamma(q) \notin G^{\text{Pl}} \quad (6.26)$$

$$b_q^{\max} = \Pi_{\gamma(q)}^{\max} \quad \gamma(q) \in G^{\text{Pl}} \quad (6.27)$$

6.2.5 Compatibility with Vivado HLS

Vivado HLS imposes two additional restrictions on the nesting of pipelined graphs.

$$O_g^{\text{Lo}} = \emptyset \quad \forall g \in G^{\text{Pl}} \quad (6.28)$$

$$b_{\sigma(i)} \mid \Pi_g \quad \forall i \in O_g^{\text{Ca}}, \forall g \in G^{\text{Pl}} \quad (6.29)$$

First, pipelined loops cannot contain other loops (6.28). Secondly, the blocking times of all function operator types used in a pipelined graph must divide the graph's Π (6.29). Note that all operator types predefined in Vivado HLS' library are fully pipelined, i.e. they have a blocking time of 1.

In the implementation of SKYCASTLE, we handle the chaining of combinatorial operations and the accesses to on-chip memory and AXI4 ports in a way that faithfully reproduces Vivado HLS' behaviour. As a concession to the clarity of this chapter, these implementation details are omitted here.

6.3 EXTENSIONS FOR MOOVAC

As a preparation for the definition of the SKYCASTLE formulation in Section 6.4.2, we present modifications to the underlying MoovAC formulation in the context of a single graph g , and call the result mMoovAC (“modified MoovAC”).

6.3.1 Alternative Modulo Decomposition

When we made the Π a decision variable in the MoovAC-I formulation in Section 3.1.2, we faced a product of decision variables in the modulo decomposition, and thus needed to linearise constraints (3.10). We chose to enumerate the possible values of the y variables then (3.19).

However, recall from Section 5.3.3 that we encoded dynamic lower bounds for the interaction of the Π and the operator allocation (5.20) with the help of binary variables π^x that indicate whether the Π has a particular value x . It is apparent that we can use these variables to linearise constraints (3.10) in an alternative way, by imposing one constraint per Π value, instead of one constraint per value of an operation’s y variable (6.30). Constraints (6.31) ensure that the π variables are assigned properly, and lastly, (6.32) are the domain constraints for the new variables.

$$\pi_g^x = 1 \rightarrow t_i = y_i \cdot x + m_i \quad \forall x \in [\Pi_g^{\min}, \Pi_g^{\max}] \quad \forall i \in O_g^q, \forall q \in Q^{\text{Sh}} \quad (6.30)$$

$$\sum_{\Pi_g^{\min} \leq x \leq \Pi_g^{\max}} \pi_g^x = 1 \quad \sum_{\Pi_g^{\min} \leq x \leq \Pi_g^{\max}} x \cdot \pi_g^x = \Pi_g \quad (6.31)$$

$$\pi_g^x \in \{0, 1\} \quad \forall x \in [\Pi_g^{\min}, \Pi_g^{\max}] \quad (6.32)$$

6.3.2 Alternative Operator Constraints

In the MoovAC formulation, linear constraints (3.3)–(3.9) model the operator constraints (2.2). In order to ensure that no more than a_q instances of a shared operator type $q \in Q^{\text{Sh}}$ are used at any time by the operations of a graph $g \in G$, we require that each pair of operations competing for the same operator type is either scheduled to different congruence classes (modulo Π), bound to different operators, or both.

In this section, we perform two modifications: first, we again drop the explicit computation of a binding between the operations and operator instances, and instead resort to counting *conflicts* across all pairs of operations using the same operator type. We already used this idea, which originates from the work of Šůcha and Hanzálek [86], albeit in a more ad-hoc manner, in Chapter 5. Secondly, we propose a formulation approach to detect conflicts between pairs of operations which supports general, and even variable, blocking times.

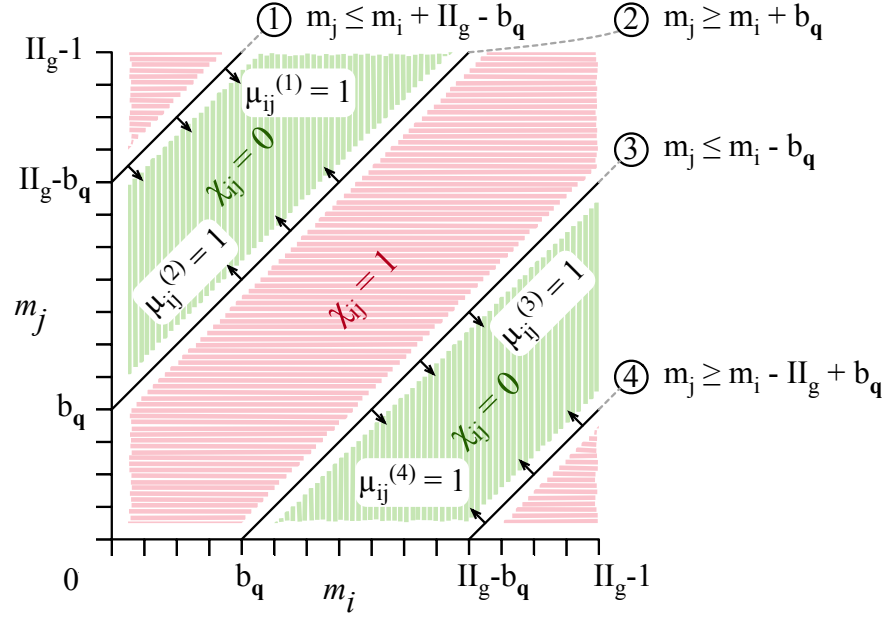


Figure 6.3: Interaction of the m , μ and χ variables in the alternative modelling of the operator constraints

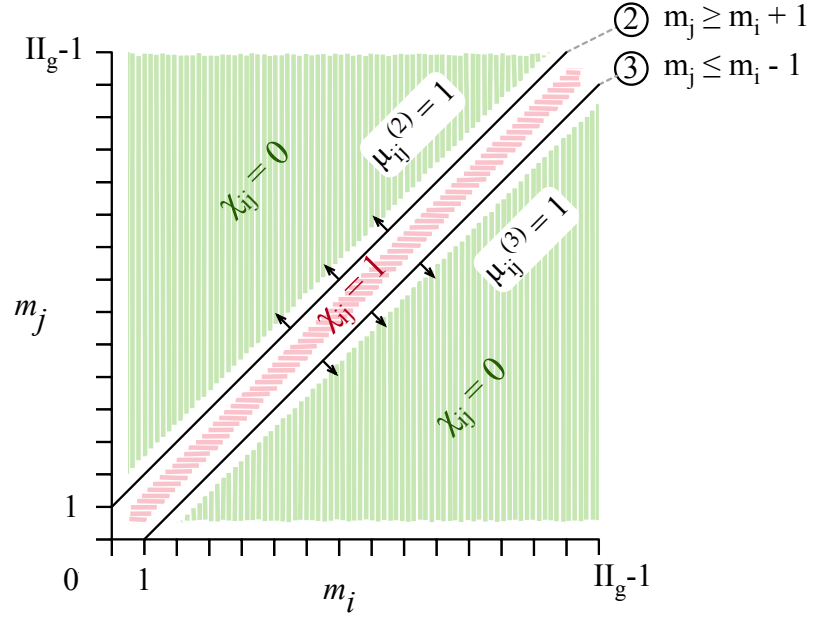


Figure 6.4: Interaction of the m , μ and χ variables in the alternative modelling of the operator constraints, simplified for fully-pipelined operator types with a blocking time of 1

In the following paragraphs, we will reason about pairs of operations $i, j \in O_g^q$ using the same shared operator type $q \in Q^{Sh}$. We assume the presence of an arbitrary total order $<$ among these operations, which lets us denote subsets of *unique* pairs, e.g. with $i < j$.

OPERATOR CONSTRAINTS VIA CONFLICT COUNTING We say two operations i, j are in conflict, indicated by a newly introduced binary variable $\chi_{ij} = 1$, if they cannot use the same q -instance due to overlapping blocking times. As this relation is symmetric, we only encode it for $i < j$. (6.33) express that we need to allocate at least one q -instance more than the maximum number of conflicts any operation is part of, as both operation i and all conflicting operations require access to a q -instance. (6.34) are the domain constraints for the new variables.

$$\sum_{j \in O_g^q: i < j} \chi_{ij} + \sum_{j \in O_g^q: i > j} \chi_{ji} \leq a_q - 1 \quad \forall i \in O_g^q \quad (6.33)$$

$$\chi_{ij} \in \{0, 1\} \quad \forall i, j \in O_g^q : i < j \quad (6.34)$$

CONFLICT DETECTION WITH VARIABLE BLOCKING TIMES Figure 6.3 illustrates the space of possible congruence class assignments, m_i and m_j , for a unique pair of operations i, j . The green areas enclose non-conflicting assignments, i.e. both operations can use the same operator. Per definition, we want to express this situation as $\chi_{ij} = 0$. All other assignments in the red areas result in overlapping blocking times in either the same or adjacent iterations, which prevent i and j from sharing one q -instance. The conflict should be expressed as $\chi_{ij} = 1$.

The different areas are bounded by four inequalities between m_i and m_j , as visualised by the lines in Figure 6.3. The arrowheads indicate on which side of the line the inequality is satisfied.

Coincidentally, the figure shows the situation for $\Pi_q = 16$ and $b_q = 4$. An assignment of $m_i = 4$ and $m_j = 8$ corresponds to a point in the upper green area, enclosed by lines 1 and 2. Setting $m_i = 3$ and $m_j = 5$ obviously leads to overlapping blocking times of the operations, and represents a point in the middle area of conflict, where neither inequality 2 nor 3 are satisfied. As a last example, consider $m_i = 14$ and $m_j = 2$, which implies a point in the bottom right red area. Here, enough time steps separate both operations in the *current* iteration to have non-overlapping blocking times. However, i starts so late in the iteration that its blocking time overlaps with operation j in the *next* iteration. Hence, i and j are in conflict.

We introduce four binary variables, $\mu_{ij}^{(1)} - \mu_{ij}^{(4)}$, for each unique pair of operations, to express if a particular inequality is satisfied by the current assignment of m_i and m_j . The indicator constraints (6.35)–(6.42) bind these variables to 1 if the respective inequality is fulfilled, and to 0 if is is not.

Our implementation actually uses a manual linearisation of these constraints with tight big-M constants. This mechanical transformation is omitted here for brevity.

From Figure 6.3, it is apparent that exactly three satisfied inequalities correspond to non-conflicting assignments of m_i and m_j , whereas conflicting assignments fulfil only two inequalities. This means we can define a conflict variable χ_{ij} by simply counting the number of satisfied inequalities in (6.43). If the value of the sum is 2, χ_{ij} must be 1, otherwise the sum equals 3, and χ_{ij} is set to 0. (6.44) are the domain constraints for the new binaries.

$$\forall i, j \in O_g^q : i < j, \forall q \in Q^{\text{Sh}} :$$

$$\mu_{ij}^{(1)} = 1 \rightarrow m_j \leq m_i + \Pi_g - b_q \quad (6.35)$$

$$\mu_{ij}^{(1)} = 0 \rightarrow m_j \geq m_i + \Pi_g - b_q + 1 \quad (6.36)$$

$$\mu_{ij}^{(2)} = 1 \rightarrow m_j \geq m_i + b_q \quad (6.37)$$

$$\mu_{ij}^{(2)} = 0 \rightarrow m_j \leq m_i + b_q - 1 \quad (6.38)$$

$$\mu_{ij}^{(3)} = 1 \rightarrow m_j \leq m_i - b_q \quad (6.39)$$

$$\mu_{ij}^{(3)} = 0 \rightarrow m_j \geq m_i - b_q + 1 \quad (6.40)$$

$$\mu_{ij}^{(4)} = 1 \rightarrow m_j \geq m_i - \Pi_g + b_q \quad (6.41)$$

$$\mu_{ij}^{(4)} = 0 \rightarrow m_j \leq m_i - \Pi_g + b_q - 1 \quad (6.42)$$

$$3 - \chi_{ij} \leq \mu_{ij}^{(1)} + \mu_{ij}^{(2)} + \mu_{ij}^{(3)} + \mu_{ij}^{(4)} \leq 3 \quad (6.43)$$

$$\mu_{ij}^{(1)}, \mu_{ij}^{(2)}, \mu_{ij}^{(3)}, \mu_{ij}^{(4)} \in \{0, 1\} \quad (6.44)$$

CONFLICT DETECTION WITH SIMPLE BLOCKING TIMES Note that for the common case of $b_q = 1$, inequalities 1 and 4 are always satisfied, as illustrated in Figure 6.4. This means we can drop the variables $\mu_{ij}^{(1)}$ and $\mu_{ij}^{(4)}$, as well as their definitions, and simplify the constraints defining χ_{ij} (6.43) to the form in (6.45).

$$1 - \chi_{ij} \leq \mu_{ij}^{(2)} + \mu_{ij}^{(3)} \leq 1 \quad \forall i, j \in O_g^q : i < j, \forall q \in Q^{\text{Sh}} : b_q = 1 \quad (6.45)$$

6.4 RESOURCE-AWARE MULTI-LOOP SCHEDULER

In general, the MLS problem is not a linear optimisation problem. However, with SKYCASTLE, we propose a solution strategy using integer linear programming (ILP) that is capable of handling a realistic, non-trivial subclass of kernels, as will be demonstrated in Section 6.5. Currently, kernels need to be legal for Vivado HLS compilation, and may contain multiple, optionally pipelined loops that may call multiple pipelined functions.

Our basic idea here is to solve the problem hierarchically, one function at a time. Figure 6.5 shows the flow, and illustrates the subclass of MLS problems solvable by SKYCASTLE. First, we leverage the iterative MORAMS approach presented in Section 5.3.2 to precompute solutions for the functions called from the top-level function. Then, the SKYCASTLE ILP formulation, presented in Section 6.4.2, simultaneously

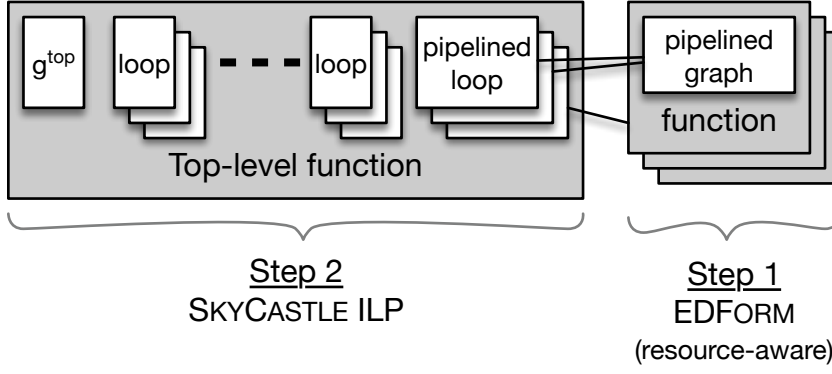


Figure 6.5: Flow of the SKYCASTLE scheduler

- schedules all loops,
- selects one of the precomputed solution for each called function, and
- determines the operator allocation

for the kernel's top-level function.

6.4.1 Precomputing Solutions

Due to the preconditions stated above, function operator types reference graphs that are pipelined, and contain neither loop nor call operations. Therefore, we can apply the iterative MORAMS approach based on the resource-aware EDFORM (see Section 5.2.1) to compute a set of solutions \mathcal{S}_q for a function operator type $q \in Q^{\text{Fu}}$ used in the top-level function.

Recall that a solution $S \in \mathcal{S}_q$ is computed by minimising the resource demand for a candidate interval Π_q^S , then fixing the resulting allocation, and finally minimising the latency. S is characterised by its interval Π^S , latency T^S , and allocation A^S . Per definition in Section 5.1.2, the set \mathcal{S}_q contains only solutions that are Pareto-optimal with regards to the Π and the resource demand. However, in order to prevent trivially infeasible MLS problem instances due to the blocking time constraints (6.29), we compute additional solutions to ensure that \mathcal{S}_q and $\mathcal{S}_{q'}$ contain solutions for the same set of Π s if q and $q' \in Q^{\text{Fu}}$ occur together in at least one pipelined graph. Alternatively, we could interpolate solutions with a compatible Π from the set of Pareto-optimal solutions by inserting empty congruence classes into the schedule, but, considering that call sites are often located inside of loops, the increased function latency would matter here, and might noticeably deteriorate the overall solution quality.

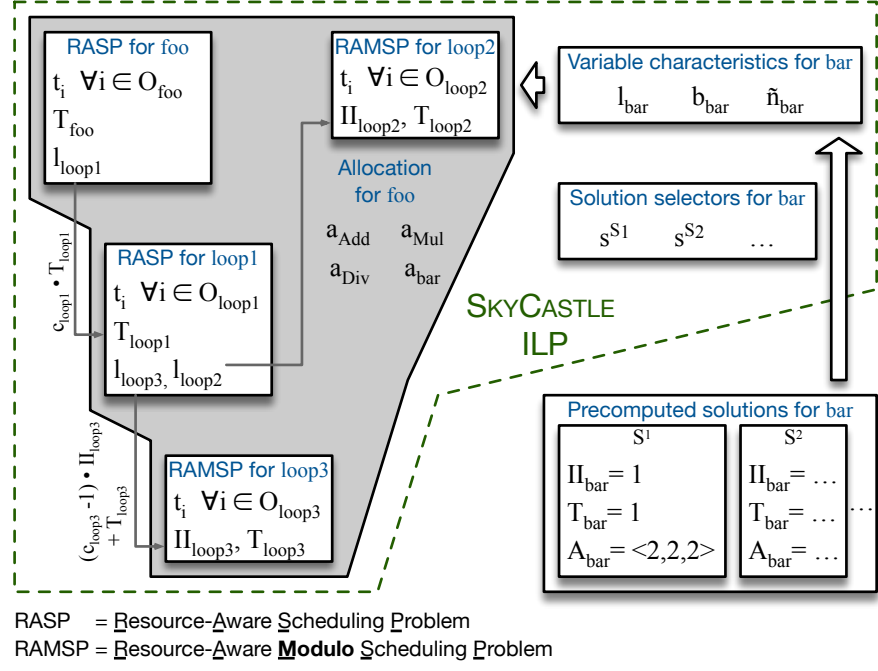


Figure 6.6: Outline of the SKYCASTLE ILP for the example instance from Figure 6.1. Not all decision variables are shown.

6.4.2 SKYCASTLE ILP Formulation

Before we present the actual formulation, let us gain an intuitive overview of its inner workings. Assume that in the example instance from Figure 6.1, the graphs loop2, loop3 and bar are pipelined, and foo and loop1 are not. Figure 6.6 outlines the components of the SKYCASTLE ILP model for this situation.

On the left-hand side, we recognise the four scheduling subproblems corresponding to the four graphs in function foo. Pipelined graphs result in resource-aware modulo scheduling problems, whereas non-pipelined graphs yield resource-aware operator-constrained scheduling problems. The problems are linked through the definition of the variable latencies of loop operations, according to the nesting structure of their loops in the kernel. The operator allocation for foo is shared by all scheduling problems. The allocation is subject to the given resource constraints in terms of the FPGA's low-level resource types (omitted in the figure).

On the right-hand side, we hint at the presence of several precomputed solutions for function bar. In the formulation, a binary decision variable indicates whether a particular solution is selected. Based on this selection, the variable latency, blocking time and resource demand for the function operator type bar is set. The scheduling problem loop2, which contains a call operation associated with this operator type, transparently handles these formerly constant values.

We assume that the kernel's control-flow is structured, and in consequence, the loop hierarchy is a tree.

The SKYCASTLE ILP formulation builds upon the resource-aware scheduling technique from Chapter 5 to handle the allocation part of the MLS problem, and uses multiple instances of the mMOOVAC formulation, as presented earlier in this chapter (Section 6.3). For the non-modulo scheduling problems, we introduce a simplified, non-modulo variant of mMOOVAC that remains exact and resource-aware.

We introduce the formulation part by part in the following paragraphs. The complete ILP consists of the objectives (6.68)–(6.69), subject to the constraints (6.70)–(6.112) and the domain constraints (6.56)–(6.67).

PARAMETERS When constructing the ILP for a function F , we consider the following model components to be parameters:

- The specification of operations, operators and resources according to Table 6.1.
- The static mapping functions of Table 6.3.
- The precomputed values of the bounds introduced in Section 6.2.4.
- The characteristics of the precomputed solutions for function operator types, as described in Section 6.4.1.
- The values of attributes, marked as either “parameter” or “derived” in Figure 6.2, as listed in (6.46)–(6.55). This includes the given resource limits (6.46). As mentioned in Equation (6.55), all predefined operators in Vivado HLS’ operator library are fully pipelined (6.50).

In order to distinguish symbols that will occur both as parameters and decision variables, e.g. an operation’s latency, we will underline the parameters from now on.

$$\underline{N}_r \in \mathbb{N} \quad \forall r \in R \quad (6.46)$$

$$\underline{c}_g \in \mathbb{N} \quad \forall g \in G_F \quad (6.47)$$

$$\underline{a}_q = \sum_{g \in G_F} |O_g^q| \quad \forall q \in Q_F^\infty \quad (6.48)$$

$$\underline{l}_q \in \mathbb{N}_0 \quad \forall q \in Q_F^{\text{Pd}} \quad (6.49)$$

$$\underline{b}_q = 1 \quad \forall q \in Q_F^{\text{Pd}} \cap Q_F^{\text{Sh}} \quad (6.50)$$

$$\underline{n}_q^r \in \mathbb{N}_0 \quad \forall q \in Q_F^{\text{Pd}}, \forall r \in R \quad (6.51)$$

$$\underline{l}_i = \underline{l}_{\sigma(i)} \quad \forall i \in O_g^{\text{No}}, \forall g \in G_F \quad (6.52)$$

$$\underline{d}_{ij} \in \mathbb{N}_0 \quad \forall (i \rightarrow j) \in E_g, \forall g \in G_F \quad (6.53)$$

$$\underline{L}_{ij} \in \mathbb{N}_0 \quad \forall (i \rightarrow j) \in E_g, \forall g \in G_F \quad (6.54)$$

$$\underline{l}_{ij} = \underline{l}_i + L_{ij} \quad \forall (i \rightarrow j) \in E_g : i \in O_g^{\text{No}}, \forall g \in G_F \quad (6.55)$$

DECISION VARIABLES Our formulation uses the decision variables defined in (6.56)–(6.62), and bounded according to Section 6.2.4, to model the corresponding components in the problem definition in Section 6.2.3.

$$T_g \in \mathbb{N}_0 \quad \forall g \in G_F \quad (6.56)$$

$$\Pi_g \in \mathbb{N}_0 \quad \forall g \in G_F^{pl} \quad (6.57)$$

$$a_q \in \mathbb{N} \quad \forall q \in Q_F^{Sh} \quad (6.58)$$

$$l_q \in \mathbb{N}_0 \quad \forall q \in Q_F^{Fu} \quad (6.59)$$

$$b_q \in \mathbb{N} \quad \forall q \in Q_F^{Fu} \quad (6.60)$$

$$t_i \in \mathbb{N}_0 \quad \forall i \in O_g, \forall g \in G_F \quad (6.61)$$

$$l_i \in \mathbb{N}_0 \quad \forall i \in O_g^{Lo} \cup O_g^{Ca}, \forall g \in G_F \quad (6.62)$$

Our formulation uses the following additional, internal decision variables.

$$s_q^S \in \{0, 1\} \quad \forall S \in \mathcal{S}_q \quad \forall q \in Q_F^{Fu} \quad (6.63)$$

$$\tilde{n}_q^r \in \mathbb{N}_0 \quad \forall q \in Q_F^{Fu}, \forall r \in R \quad (6.64)$$

$$\pi_g^x \in \{0, 1\} \quad \forall x \in [\Pi_g^{\min}, \Pi_g^{\max}], \forall g \in G_F^{pl} \quad (6.65)$$

$$y_i \in \mathbb{N}_0, m_i \in \mathbb{N}_0 \quad \forall i \in O_g : i \notin O_g^{Lo} \wedge \sigma(i) \in Q_F^{Sh} \quad (6.66)$$

$$\mu_{ij}^{(1)}, \mu_{ij}^{(2)}, \mu_{ij}^{(3)}, \mu_{ij}^{(4)}, \quad \forall i, j \in O_g^q : i < j, \forall q \in Q_F^{Sh} \quad (6.67)$$

$$\chi_{ij} \in \{0, 1\} \quad \forall g \in G_F$$

The decision variables in (6.65)–(6.67) stem from the mMOOVAC formulation of the scheduling subproblems.

(6.63) model that a precomputed solution is selected for a function operator type. (6.64) represent the accumulated resource demand for *all* allocated instances of a function operator type. (6.65) correspond to a particular value of a graph's Π . (6.66) are used to decompose the start time of an operation i in a pipelined graph g as $t_i = y_i * \Pi_g + m_i$. We call m_i the modulo slot, i.e. the congruence class modulo the graph's Π . (6.67) help to govern the maximum concurrent use of the shared operator types. As we only need to define these variables for unique pairs of operations, we assume the presence of an arbitrary total order $<$ among the operations.

OBJECTIVE We consider two objectives, and optimise lexicographically. The primary objective (6.68) is, as in the general MLS problem, to minimise the latency of the top-level graph, $T_{g^{\text{top}}}$. As a practical consideration, we seek to find the most resource-efficient solution in case multiple solutions with shortest possible latency exist. To that end, the secondary objective (6.69) is to minimise the accumulated,

weighted resource demand (cf. $f^{\text{RU}}(S)$, defined in (5.5) in Section 5.1.2). Note that the term in parentheses is equivalent to $n^r(A_F)$.

$$\min T_{g^{\text{top}}} \quad (6.68)$$

$$\min \frac{1}{|R|} \cdot \sum_{r \in R} \frac{1}{N_r} \cdot \left(\sum_{q \in Q_F^{\infty}} a_q \cdot \underline{n}_q^r + \sum_{q \in Q_F^{\text{pd}} \cap Q_F^{\text{sh}}} a_q \cdot \underline{n}_q^r + \sum_{q \in Q_F^{\text{fu}}} \tilde{n}_q^r \right) \quad (6.69)$$

RESOURCE CONSTRAINTS (6.70) model the MLS problem's resource constraints (6.16) for a resource type $r \in R$. Using the \tilde{n}_q^r -variables here avoids the product of decision variables.

$$\sum_{q \in Q_F^{\infty}} a_q \cdot \underline{n}_q^r + \sum_{q \in Q_F^{\text{pd}} \cap Q_F^{\text{sh}}} a_q \cdot \underline{n}_q^r + \sum_{q \in Q_F^{\text{fu}}} \tilde{n}_q^r \leq N_r \quad (6.70)$$

OPERATOR CONSTRAINTS In order to satisfy the operator constraints (6.15) in the MLS problem, we implement the conflict-counting mechanism discussed in Section 6.3.2. (6.71) express that we need to allocate at least one q -instance more than the maximum number of conflicts any operation is part of.

$$\sum_{j \in O_g^q: i < j} \chi_{ij} + \sum_{j \in O_g^q: i > j} \chi_{ji} \leq a_q - 1 \quad \forall i \in O_g^q, \forall g \in G_F, \forall q \in Q_F^{\text{sh}} \quad (6.71)$$

These constraints rely on the correct definition of the conflict variables χ , which we will establish separately for modulo and non-modulo subproblems.

SELECTION OF SOLUTIONS (6.72)-(6.74) are indicator constraints that derive the latency, blocking time and resource demand of a function operator type $q \in Q_F^{\text{fu}}$ from a precomputed solution (cf. Section 6.2.3). (6.75) enforce that exactly one solution is picked.

$$s_q^S = 1 \rightarrow l_q = T_{\gamma(q)}^S \quad \forall S \in \mathcal{S}_q \quad (6.72)$$

$$s_q^S = 1 \rightarrow b_q = \Pi_{\gamma(q)}^S \quad \forall S \in \mathcal{S}_q \quad (6.73)$$

$$s_q^S = 1 \rightarrow \tilde{n}_q^r = a_q \cdot n^r(A_{\varphi(\gamma(q))}^S) \quad \forall S \in \mathcal{S}_q, \forall r \in R \quad (6.74)$$

$$\sum_{S \in \mathcal{S}_q} s_q^S = 1 \quad (6.75)$$

VARIABLE LATENCIES (6.76)–(6.78) propagate the variable latencies of the loop and call operations in every graph $g \in G_F$. (6.79) and (6.80) define the graph’s latency for fixed and variable latency operations.

$$l_i = c_{\gamma(i)} \cdot T_{\gamma(i)} \quad \forall i \in O_g^{Lo} \wedge g \notin G^{Pl} \quad (6.76)$$

$$l_i = (c_{\gamma(i)} - 1) \cdot \Pi_{\gamma(i)} + T_{\gamma(i)} \quad \forall i \in O_g^{Lo} \wedge g \in G^{Pl} \quad (6.77)$$

$$l_i = l_{\sigma(i)} \quad \forall i \in O_g^{Ca} \quad (6.78)$$

$$t_i + l_i \leq T_g \quad \forall i \in O_g^{No} \quad (6.79)$$

$$t_i + l_i \leq T_g \quad \forall i \in O_g^{Lo} \cup O_g^{Ca} \quad (6.80)$$

MODULO SCHEDULING PROBLEMS Every pipelined graph $g \in G_F^{Pl}$ implies one modulo scheduling problem, for which we instantiate the constraints of the mMoovac formulation (Section 6.3).

We adopt the already linear precedence constraints (6.14) in the MLS problem as (6.81) and (6.82), again differentiating fixed and variable latency operations.

$$t_i + l_{ij} \leq t_j + d_{ij} \cdot \Pi_g \quad \forall (i \rightarrow j) \in E_g : i \in O_g^{No} \quad (6.81)$$

$$t_i + l_i + L_{ij} \leq t_j + d_{ij} \cdot \Pi_g \quad \forall (i \rightarrow j) \in E_g : i \notin O_g^{No} \quad (6.82)$$

(6.83) define the binary variable π_g^x to be 1 iff the value of Π_g is x . Using these variables and multiple indicator constraints (6.84), we linearise the decomposition of an operation’s start time into a multiple of the Π and the modulo slot.

$$\sum_{\Pi_g^{min} \leq x \leq \Pi_g^{max}} \pi_g^x = 1 \quad \sum_{\Pi_g^{min} \leq x \leq \Pi_g^{max}} x \cdot \pi_g^x = \Pi_g \quad (6.83)$$

$$\pi_g^x = 1 \rightarrow t_i = y_i \cdot x + m_i \quad \forall x \in [\Pi_g^{min}, \Pi_g^{max}] \quad \forall i \in O_g^q, \forall q \in Q_F^{Sh} \quad (6.84)$$

(6.85) model Vivado HLS’ constraint (6.29) regarding the blocking times of function operator types: For each solution S , we determine a set of viable Π s for g that restrict the feasible values for Π_g if S is selected.

$$\sum_{x \in [\Pi_g^{min}, \Pi_g^{max}] : \Pi_{\gamma(q)}^S \mid x} \pi_g^x \geq s_q^S \quad \forall S \in \mathcal{S}_q, \forall q \in Q_F^{Fu} \quad (6.85)$$

In order to establish the correct behaviour for the conflict variables used in the operator constraints (6.71), we instantiate the constraints

from Section 6.3.2 as follows: For function operator types, we use the most generic form as in (6.86)–(6.94).

$$\forall i, j \in O_g^q : i < j, \forall q \in Q_F^{\text{Fu}} :$$

$$\mu_{ij}^{(1)} = 1 \rightarrow m_j \leq m_i + \Pi_g - b_q \quad (6.86)$$

$$\mu_{ij}^{(1)} = 0 \rightarrow m_j \geq m_i + \Pi_g - b_q + 1 \quad (6.87)$$

$$\mu_{ij}^{(2)} = 1 \rightarrow m_j \geq m_i + b_q \quad (6.88)$$

$$\mu_{ij}^{(2)} = 0 \rightarrow m_j \leq m_i + b_q - 1 \quad (6.89)$$

$$\mu_{ij}^{(3)} = 1 \rightarrow m_j \leq m_i - b_q \quad (6.90)$$

$$\mu_{ij}^{(3)} = 0 \rightarrow m_j \geq m_i - b_q + 1 \quad (6.91)$$

$$\mu_{ij}^{(4)} = 1 \rightarrow m_j \geq m_i - \Pi_g + b_q \quad (6.92)$$

$$\mu_{ij}^{(4)} = 0 \rightarrow m_j \leq m_i - \Pi_g + b_q - 1 \quad (6.93)$$

$$3 - \chi_{ij} \leq \mu_{ij}^{(1)} + \mu_{ij}^{(2)} + \mu_{ij}^{(3)} + \mu_{ij}^{(4)} \leq 3 \quad (6.94)$$

For the remaining predefined and shared operator types, we use the simpler form (6.95)–(6.99).

$$\forall i, j \in O_g^q : i < j, \forall q \in Q_F^{\text{Sh}} \cap Q_F^{\text{Pd}} :$$

$$\mu_{ij}^{(2)} = 1 \rightarrow m_j \geq m_i + 1 \quad (6.95)$$

$$\mu_{ij}^{(2)} = 0 \rightarrow m_j \leq m_i \quad (6.96)$$

$$\mu_{ij}^{(3)} = 1 \rightarrow m_j \leq m_i - 1 \quad (6.97)$$

$$\mu_{ij}^{(3)} = 0 \rightarrow m_j \geq m_i \quad (6.98)$$

$$1 - \chi_{ij} \leq \mu_{ij}^{(2)} + \mu_{ij}^{(3)} \leq 1 \quad (6.99)$$

The following bounds help to restrict the ILP solution space further: (6.100) encode a slightly weaker form of the dynamic lower bound (5.20) from Section 5.3.3 for the shared operator types. (6.101) mandate that Π_g is greater or equal to the longest blocking time of any selected function operator type.

$$x \cdot a_q \geq \pi_g^x \cdot |O_g^q| \cdot b_q^{\min} \quad \forall x \in [\Pi_g^{\min}, \Pi_g^{\max}], \forall q \in Q_F^{\text{Sh}} \quad (6.100)$$

$$\Pi_g \geq s_q^S \cdot \Pi_{\gamma(q)}^S \quad \forall S \in \mathcal{S}_q, \forall q \in Q_F^{\text{Fu}} \quad (6.101)$$

NON-MODULO SCHEDULING PROBLEMS Every non-pipelined graph $g \in G_F \setminus G_F^{\text{Pl}}$ imposes a resource-constrained, or rather, operator-constrained, scheduling problem, for which we instantiate a non-modulo variant of the mMOOVAC formulation as follows.

We again directly include the precedence constraints (6.14) from the MLS problem as (6.102) and (6.103).

$$t_i + l_{ij} \leq t_j \quad \forall (i \rightarrow j) \in E_g : i \in O_g^{\text{No}} \quad (6.102)$$

$$t_i + l_i + L_{ij} \leq t_j \quad \forall (i \rightarrow j) \in E_g : i \notin O_g^{\text{No}} \quad (6.103)$$

With (6.104)–(6.112), we basically use the same modelling technique as for the modulo scheduling problems to define the conflict variables

for two operations i and j . The only differences are that we consider the operations' start times instead of their congruence classes, and can discard inequalities 1 and 4, as the overlapping of iterations is not a concern here.

$$1 - \chi_{ij} \leq \mu_{ij}^{(2)} + \mu_{ij}^{(3)} \leq 1 \quad \forall i, j \in O_g^q : i < j, \forall q \in Q_F^{\text{Sh}} \quad (6.104)$$

$$\forall i, j \in O_g^q : i < j, \forall q \in Q_F^{\text{Fu}} :$$

$$\mu_{ij}^{(2)} = 1 \rightarrow t_j \geq t_i + b_q \quad (6.105)$$

$$\mu_{ij}^{(2)} = 0 \rightarrow t_j \leq t_i + b_q - 1 \quad (6.106)$$

$$\mu_{ij}^{(3)} = 1 \rightarrow t_j \leq t_i - b_q \quad (6.107)$$

$$\mu_{ij}^{(3)} = 0 \rightarrow t_j \geq t_i - b_q + 1 \quad (6.108)$$

$$\forall i, j \in O_g^q : i < j, \forall q \in Q_F^{\text{Sh}} \cap Q_F^{\text{Pd}} :$$

$$\mu_{ij}^{(2)} = 1 \rightarrow t_j \geq t_i + 1 \quad (6.109)$$

$$\mu_{ij}^{(2)} = 0 \rightarrow t_j \leq t_i \quad (6.110)$$

$$\mu_{ij}^{(3)} = 1 \rightarrow t_j \leq t_i - 1 \quad (6.111)$$

$$\mu_{ij}^{(3)} = 0 \rightarrow t_j \geq t_i \quad (6.112)$$

6.5 CASE STUDIES

6.5.1 Kernels

In the following sections, we introduce our three kernels SPN, FFT and LULESH. Figures 6.7 to 6.9 illustrate the nesting structure of the underlying MLS problems. The dependence graphs are not shown, however we outline which shared operator types and memory ports are used, and how many users they have.

SPN Sum-Product Networks (SPNs) [72] are a relatively young type of deep machine-learning models from the class of Probabilistic Graphical Models (PGMs), for which inference has been successfully accelerated in prior work [83]. An SPN captures the joint probability distribution of its input variables in the form of a directed, acyclic graph. The graph comprises three different kinds of nodes: Weighted sums, products and, as leaf nodes, univariate distributions, which can for example be modelled as histograms [60].

The SPN kernel, as outlined in Listing 6.1, combines three inference processes for an example SPN from the NIPS corpus [22]. Assuming given values for the input variables i_1 to i_4 , we seek to find the most probable explanation for the missing input feature i_5 in a first step. For this purpose, we iterate over all 256 possible values of i_5 and evaluate the SPN (loop MPE, which has been manually unrolled by a factor 4 and is pipelined). After that, we marginalise [72] input variable i_1 , and compute a conditional probability using the most probable explanation

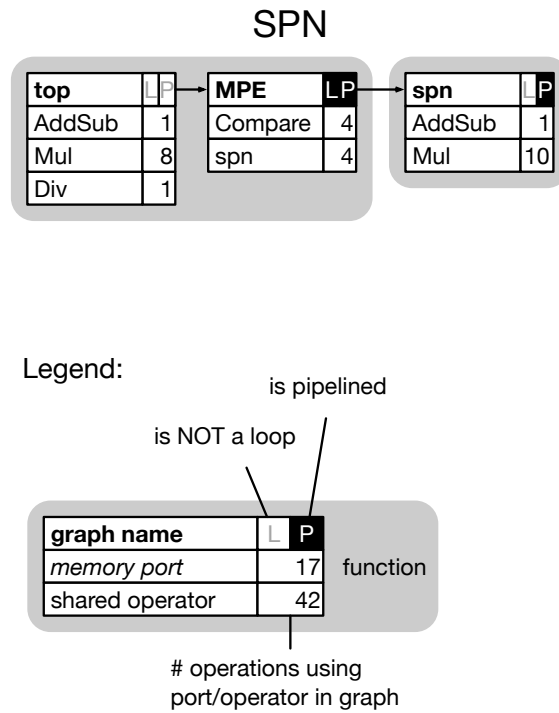


Figure 6.7: MLS problem structure for SPN kernel, and legend

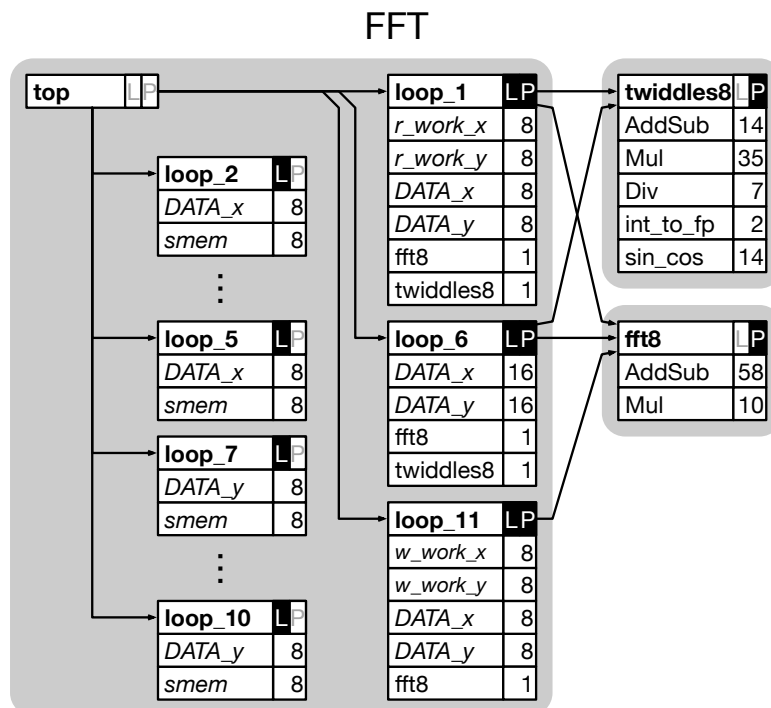


Figure 6.8: MLS problem structure for FFT kernel

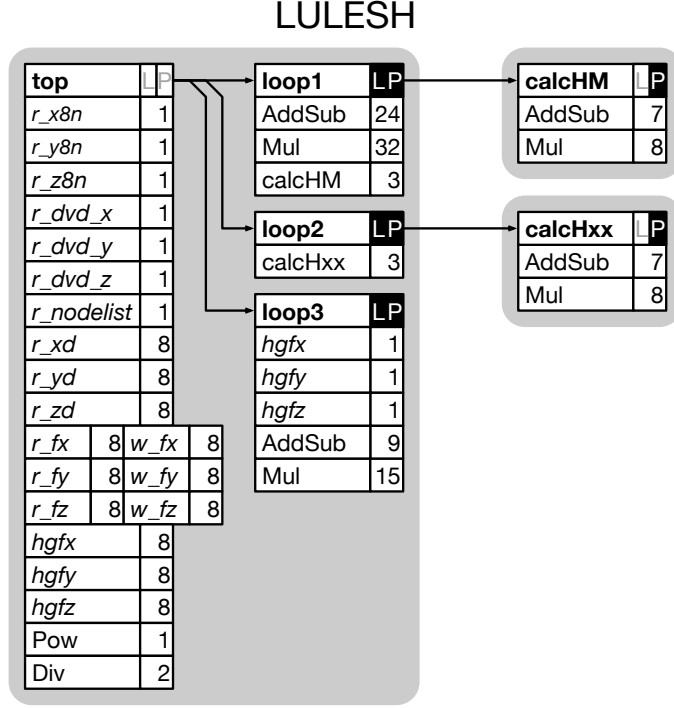


Figure 6.9: MLS problem structure for LULESH kernel

for i_5 . SPN is a small kernel, but not memory-bound, and therefore is well suited to demonstrate the benefits of accelerator replication.

FFT Our second kernel, FFT, is the fft/transpose benchmark from MachSuite [75]. One invocation processes a 512 byte chunk of input. We wrapped the FFT8 macro in a function `fft8`, and disabled inlining for it as well as for the `twiddles8` function. The top-level function contains 11 loops in total, out of which three loops are pipelined and call either one or both of the functions. FFT therefore challenges the scheduler to obey the blocking time constraints (6.29).

LULESH Our LULESH kernel represents one iteration in the `CalcFBHourglassForceForElems` function from the serial version of the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics proxy application [45]. In order to make the code compatible with Vivado HLS, we hardcoded dynamic array sizes to the default values in the application. We replaced the cubic root function by the power of $\frac{1}{3}$, as not even one `cbrt` operator would fit on the XC7Z020 device together with the minimal allocation of the other operator types. In order to obtain a best-effort HLS version of the code, we inlined the `CalcElemFBHourglassForce` function and restructured the loops in it. Additionally, we extracted common functionality into new functions `calcHM` and `calcHxx`. The resulting three loops and two functions

are all pipelined. This kernel contains the most complex allocation problem of our case studies, as non-trivial computations in the loops compete with the variable allocation and solution selection of function operators.

6.5.2 Experimental Setup

Our current SKYCASTLE implementation considers Look-Up Tables (LUTs), Flip-Flops (FFs), Block RAMs (BRAMs) and Digital Signal Processing Blocks (DSPs), i.e. the typical low-level resource types on Xilinx devices. We target the ZedBoard (XC7Z020: 53,200 LUT; 106,400 FF; 280 BRAM; 220 DSP) at 100 MHz and the VCU108 evaluation board (XCVU095: 537,600 LUT; 1,075,200 FF; 3,456 BRAM; 768 DSP) at 200 MHz, and compose bitstreams for complete SoC designs, comprised of one or more accelerators, with TaPaSCo 2019.6 [50] and Vivado 2018.3.

In order to accommodate TaPaSCo's SoC template, as well as to give the logic synthesis tools some headroom, we make 85% (ZedBoard) respectively 70% (VCU108, more complex template due to PCIe interface) of the resources available to the allocation of operators during scheduling. The MLS problem instances are extracted from Vivado HLS 2018.3 operating with medium effort levels for scheduling and binding, and using a target cycle time of 4 ns (VCU108) or 8 ns (ZedBoard) without clock uncertainty. We marked loops and functions for pipelining without specifying the II.

SKYCASTLE uses the Gurobi 8.1 ILP solver, which was allowed to use up to 8 threads and 16 GB RAM per kernel. The experiments were performed on 2×12 -core Intel Xeon E5-2680 v3 systems running at 2.8 GHz with 64 GiB RAM. We set time limits of 15 min for the primary objective (minimisation of the latency, (6.68)), and 5 min for the secondary objective (subsequent minimisation of the resource demand, (6.69)). The same limits were in place for the computation of the solutions for the function operator types, but recall that the primary objective is to minimise the resource demand here. If the solver is unable to prove optimality within the time limit, we accept the feasible solution, and record the optimality gap, relative to the solver's best lower bound.

For the larger VCU108 board, we experiment with different *replication factors* $1 \leq \rho \leq 8$, and set the resource limits passed to the scheduler according to (6.113).

$$N_r = \frac{\text{available r-elements on FPGA} - \text{headroom}}{\rho} \quad \forall r \in R \quad (6.113)$$

On the much smaller ZedBoard, we only used $\rho = 1$.

For the configurations labeled "SC- $x\rho$ ", as referenced in the following discussion, we computed a solution adhering to these resource

constraints with SKYCASTLE, emitted pipeline and allocation directives accordingly, and ran Vivado HLS again with them. The configuration labeled “VHLS” denotes the baseline maximum performance that Vivado HLS constructs without the SKYCASTLE optimisation.

6.5.3 Results

Table 6.5 summarises the high- and low-level synthesis results. Column “Latency” shows the latency (in cycles) of one activation of the kernel’s top-level function, as reported by Vivado HLS. SkyCastle’s estimation of the Vivado HLS’ cycle count (not shown) is very precise and differs by at most 2.4%, which shows that we model enough of Vivado HLS’ scheduling peculiarities to meaningfully tackle the problem. The next column “Util.” extracts the utilisation of DSP slices, which were always the scarcest resource type in our evaluation, from the HLS report. SkyCastle’s estimation of DSP slices is almost perfect, and is off by at most three, because Vivado HLS appears to ignore the allocation directives for the combined floating-point ADD/SUB core in some situations. The estimation error for LUTs and FFs is below 10%, but ranges up to 70% for BRAMs. The reason for the high deviation in the latter case is that the majority of BRAM is used by components that are not operators themselves, and thus do not occur in the MLS problem. However, as mentioned above, the BRAM utilisation was never crucial in our experiments.

The remaining columns characterise results of composing a bit-stream comprised of “# Acc.”-many accelerators.

Most importantly, column “Freq.” shows SkyCastle accomplished its mission: While neither FFT nor LULESH fit on the devices with the default VHLS flow, we computed synthesisable configurations for up to four replica. For both kernels, the scheduler determined $\rho = 5$ to be infeasible even with maximum resource sharing. SPN does fit once on the larger device with the default flow, but this configuration cannot be replicated. Again, all SKYCASTLE configurations yielded working *multi-accelerator* designs. Note that the SC-x1 configuration for the VCU108 is slower and uses less resources than the VHLS configuration. This is because Vivado HLS violates the target cycle time slightly (but without consequences in this particular situation), whereas SKYCASTLE strictly obeys this user constraint and computes a more conservative solution.

The last column, “Throughp.” states the theoretical throughput achievable with each multi-accelerator design, calculated as

$\left(\frac{\# \text{ Acc.}}{\text{Latency}} \cdot \text{Freq.} \right)$. When viewed together with the column “Latency”, the benefits of scheduling for better replicability become apparent. For all three kernels, SC-x2 already yields a *better* throughput than the *maximum performance* VHLS configuration. SPN reaches its theoretical peak performance with a 7-way accelerator, whereas the other kernels profit from any additional replication.

The biggest challenge for SKYCASTLE was to schedule LULESH for the 2-way accelerator design. The feasible solution had an optimality gap of 5.2% after optimising the primary objective for 15 min, and a gap of 0.09% remained after 5 min spent on the secondary objective. In all other cases, the ILP solver either returned optimal solutions, or the remaining optimality gap was in the same ballpark as the inaccuracies in the latency estimation. Note that we computed the solutions for the function operator types only once per exploration of the replication factors. Altogether, using the aforementioned time limits, the entire process took 51 minutes for LULESH targeting the VCU108, and well below 20 minutes for the other configurations.

6.6 CHAPTER SUMMARY

We formalised a novel, general scheduling and allocation model for the common problem of minimising the latency of a complex HLS kernel subject to low-level resource constraints. This model is the foundation for SKYCASTLE, our proposed resource-aware multi-loop scheduler, which currently handles a subset of kernels compatible with Xilinx Vivado HLS.

In the future, we plan to investigate improvements or alternatives to the precomputation of solutions for the function operators, which we believe will allow us to treat an arbitrary nesting structure in a uniform way. Also, our approach would benefit tremendously from a vendor-supported, high-level synthesis counterpart to the XDL interface [62], as we currently can only feed the II and the operator allocation back to Vivado HLS in the form of directives. Should such an interface become available in the future, SkyCastle could be easily adapted to replace the built-in scheduler.

Table 6.5: Scheduling and system composition results

Kernel	Board	Config	HLS		Composition		
			Latency	Util.	#	Freq.	Throughp.
			[Cyc.]	[%]	Acc.	[MHz]	[1/μs, theo.]
SPN	ZedBoard	VHLS	175	226.8	1	<i>failed</i>	
		SC-x1	366	76.8	1	100	0.55
	VCU108	VHLS	212	65.0	1	200	0.94
					2	<i>failed</i>	
		SC-x1	277	36.3	1	200	0.72
		SC-x2	278	33.5	2	200	1.44
		SC-x3	402	22.0	3	200	1.49
		SC-x4	408	16.9	4	200	1.96
		SC-x5	659	12.6	5	200	1.52
		SC-x6	663	10.8	6	200	1.81
SC-x7	665	9.4	7	200	2.11		
SC-x8	787	8.3	8	200	2.03		
FFT	ZedBoard	VHLS	4479	883.2	1	<i>failed</i>	
		SC-x1	5534	82.7	1	100	0.02
	VCU108	VHLS	4682	247.5	1	<i>failed</i>	
		SC-x1	4700	64.7	1	155	0.03
		SC-x2	4918	34.2	2	159	0.06
		SC-x3	5721	23.3	3	194	0.10
		SC-x4	6641	17.3	4	187	0.11
LULESH	ZedBoard	VHLS	533	528.2	1	<i>failed</i>	
		SC-x1	656	82.7	1	100	0.15
	VCU108	VHLS	610	150.4	1	<i>failed</i>	
		SC-x1	622	69.3	1	200	0.32
		SC-x2	681	34.4	2	200	0.59
		SC-x3	745	22.8	3	200	0.81
		SC-x4	863	17.7	4	200	0.93
Target frequencies: 100 MHz (ZedBoard), 200 MHz (VCU108)							

CONCLUSION

We conclude this thesis by summarising the key contributions and insights, and outline future research directions.

7.1 ON THE PRACTICALITY OF ILP-BASED MODULO SCHEDULING

While it certainly required a bit of stubbornness in the beginning, we believe to have found strong arguments in favour of **ILP**-based modulo scheduling.

QUALITY OF SOLUTIONS With the exact, ILP-based formulations investigated in this thesis, an off-the-shelf ILP solver was able to compute provably optimal solutions for the majority of instances from our representative data set of scheduling problems from three different **HLS** environments.

DETECTION OF INFEASIBILITY In contrast to exact approaches, heuristic modulo schedulers cannot detect the infeasibility of a candidate II. Instead, they can only run out of time or backtracking steps [7, 74] before giving up and trying the next II.

FINE-GRANULAR CONTROL OVER RUNTIME The **MSP** is without doubt a hard combinatorial problem, and our evaluations always contained a few instances that marked the end of scalability for the exact schedulers. However, by taking measures as simple as setting a time limit, we can control the maximum time spend per scheduling problem in practice, and resort to heuristic alternatives if a particular instances is too challenging for the ILP-based scheduler. Our evaluation suggests that short time limits between 1 and 5 minutes are sufficient, which seems acceptable considering the rest of the **FPGA** synthesis process is usually measured in *hours* of run time.

Our proposed solution strategy for the MOOVAC-I formulation gives users even more control, because they can specify *separately* how much time they want to expend on the II minimisation and the secondary objective.

SOFT FAILURE MODE To pick up the last point, an additional benefit of ILP-based scheduling is that even if a given time budget is depleted without finding an optimal solution, it is often the case that a feasible solution is present. Recall that the ILP-solver maintains a gap between a lower bound for the optimal objective value, and the incumbent's

The “incumbent” is the currently best available integer solution.

objective value. Based on the gap value, a user can decide whether they want to give the solver just a little more time, or abort the process. Scheduling approaches that incrementally legalise a solution, regardless whether they operate heuristically [7] or exactly [17], do not exhibit a similarly soft failure mode.

PORTABILITY An ILP formulation is a very compact representation of a scheduling approach, and often fits on a single page in a conference paper. Schedulers defined in other frameworks might only be reported incompletely, or require significantly more effort to be reproduced, for example, to re-implement the problem-specific solving algorithm in the CP framework [6], or to orchestrate both an LP and SAT solver working in unison to produce a schedule [17].

7.2 ON THE FLEXIBILITY OF THE MOOVAC FORMULATION

The experimental evaluation conducted in the course of this thesis yielded no clear winner among the ILP formulations MOOVAC, EDFORM and SHFORM regarding the scheduling runtimes.

However, the modelling approach underlying Moovac turned out to be the most adaptable to different requirements, as is apparent in Table 7.1, which summarises the capabilities of the different formulations. “Variable II” means the II can be made a decision variable. Formulations with “ $b_q > 1$ ” support operators that are not fully-pipelined. The column “Binding” denotes that a formulation produces a binding between operations and operators. Lastly, the “RA” indicates that all approaches can be extended to resource awareness in the sense of Chapter 5.

Across Moovac and mMoovac, we presented two different linearisations of the modulo decomposition (3.10), made possible because the decomposition is decoupled from the handling of the dependence edges (3.2). The Moovac operator constraints yield a binding, but only support fully-pipelined operator types. In the mMoovac extensions, we added support for arbitrary and even variable blocking times. While not implemented during this thesis, the latter modelling can also be extended to compute a binding.

The II cannot be made a variable in the EDFORM, because the number of variables and constraints depend on its value. In the general-processing-time-variant of the SHFORM, constraints (12)–(13) in [86] prevent a Moovac-I-style linearisation, because the II (“w” in their notation) is not a big-M constant there.

7.3 ON NEW DESIGN METHODS FOR HARDWARE ACCELERATORS

The ILP modelling techniques discussed in this thesis matter beyond their theoretical appeal, because they open the door for a new, more

Table 7.1: Formulation capabilities

	Variable II	$b_q > 1$	Binding	RA
EDFORM (Section 2.8.1, [26])	-	✓	(✓) [29]	✓
SHFORM (Section 2.8.2, [86])	(✓)	-	-	✓
Formulation with general processing time from [86]	-	✓	✓	✓
MOOVAC (Section 3.1)	✓	-	✓	✓
MMOOVAC (Section 6.3)	✓	✓	(✓)	✓

✓: supported (✓): possible, not implemented here -: not supported

automated HLS design method. The SKYCASTLE multi-loop scheduler is a culmination of the insights gathered during the course of this thesis. Its main feature is its ability to answer the question, “what is the fastest microarchitecture for a kernel that fits within the given resource constraints?” – a question that state-of-the-art commercial HLS tools cannot answer automatically.

7.4 OUTLOOK

Based on the insights gained in this thesis, we identify the following research avenues for future work.

A QUANTITATIVE SURVEY OF MODULO SCHEDULING APPROACHES
 To the best of our knowledge, there is no comprehensive experimental evaluation of more than a few modulo scheduling approaches under the same conditions. To that end, it would be interesting to reimplement the most promising approaches from the last three decades in a common framework, and expose them to a variety of scheduling problems. First steps in this direction have already been made: HatScheT [80], the holistic and tweakable scheduling toolkit, aims to be a collection of scheduler implementations (and already supports all schedulers discussed in this thesis), as well as a friendly environment to foster research into new approaches. GeMS, a generator for modulo scheduling problems, randomly constructs scheduling problems according to a rich set of parameters, including a mode that constructs infeasible MSP instances [69]. GeMS therefore could be used to augment an existing benchmark set with synthetic instances to cover sparsely covered combinations of instance characteristics.

ORACLE Once a data set as outlined in the previous paragraph is available, we envision the design of an oracle, capable of selecting the

most promising scheduling approach for a given problem instance. For example, “easy” instances could be solved to optimality with an exact scheduler, whereas instances suspected to be “hard” could be delegated to a heuristic algorithm. This could be based on a statistical analysis, or using machine learning. In any case, coming up with a concise definition of “easy” and “hard” in this context is an open research problem.

COMPONENT SELECTION AND COMPILER TRANSFORMATIONS

The SKYCASTLE ILP formulation supports operator types with variable latencies, blocking times and resource demands. Currently, we use this mechanism to let the ILP select a particular solution for a pipelined function globally, i.e. for all users of the operator type. We envision two modifications that would provide an additional practical benefit.

First, the selection could be made locally, for each individual operator. For example, assume that the operator library provides two implementations of a floating-point multiplier: a slower core using only LUT resources, and a faster one using the scarcer DSPs slices. A combined scheduling and allocation approach such as SKYCASTLE could then balance the use of the FPGA resources. Sun, Wirthlin and Neuendorffer [87] tackled this problem using heuristic algorithms – with the techniques presented in this thesis, it could be solved optimally.

Secondly, many compiler transformations can be expressed on graphical IRs [85]. We could use the aforementioned selection mechanism to decide whether to apply a particular transformation during scheduling. For example, in [64], we investigated domain-specific transformations for biomedical simulation models. A common pattern in this context is $e^{x+c_1} \cdot c_2$, for a variable x and constants c_1, c_2 . The expression can be constant-folded into $e^x \cdot (e^{c_1} \cdot c_2)$, or $e^{x+(c_1+\ln c_2)}$, saving either a floating-point addition or multiplication. Within a combined scheduling and allocation problem, one form can be preferable, e.g. the second form in case the allocated multipliers are busy, but unused adders are available.

INTEGRATION INTO OPENCL FLOWS We mentioned in Chapter 1 that the FPGA community whole-heartedly adopted the OpenCL ecosystem [47] to construct application accelerators for real-world problems. A selling point for FPGA engineers is that they can describe the application as a set of OpenCL kernels, and use so-called channels to connect the kernels to form larger pipelines. This area would benefit greatly from an *automated* approach to distribute the FPGA resources to the individual kernels in a way that maximises the overall performance. We believe SKYCASTLE could be extended to handle this situation, as after all, such a pipeline would map to a multi-loop scheduling problem with an allocation domain per kernel.

Unfortunately, both Intel's and Xilinx' OpenCL flows are proprietary, and offer no interface to offload the HLS core steps to an external tool yet.

BIBLIOGRAPHY

- [1] Erik R. Altman and Guang R. Gao. ‘Optimal Modulo Scheduling Through Enumeration’. In: *International Journal of Parallel Programming* 26.2 (1998), pp. 313–344. DOI: [10.1023/A:1018742213548](https://doi.org/10.1023/A:1018742213548).
- [2] Erik R. Altman, Ramaswamy Govindarajan and Guang R. Gao. ‘Scheduling and Mapping: Software Pipelining in the Presence of Structural Hazards’. In: *Proceedings of the ACM SIGPLAN’95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, USA, June 18-21, 1995*. Ed. by David W. Wall. ACM, 1995, pp. 139–150. DOI: [10.1145/207110.207128](https://doi.org/10.1145/207110.207128).
- [3] Amazon EC2 F1 instances. 2019. URL: <https://aws.amazon.com/de/ec2/instance-types/f1/>.
- [4] Maria Ayala and Christian Artigues. *On integer linear programming formulations for the resource-constrained modulo scheduling problem*. Tech. rep. LAAS no. 10393. Archive ouverte HAL, 2010. URL: <https://hal.archives-ouvertes.fr/hal-00538821>.
- [5] Peter J. Billington, John O. McClain and L. Joseph Thomas. ‘Mathematical Programming Approaches to Capacity-Constrained MRP Systems: Review, Formulation and Problem Reduction’. In: *Management Science* 29.10 (1983), pp. 1126–1141. DOI: [10.1287/mnsc.29.10.1126](https://doi.org/10.1287/mnsc.29.10.1126).
- [6] Alessio Bonfietti, Michele Lombardi, Luca Benini and Michela Milano. ‘CROSS cyclic resource-constrained scheduling solver’. In: *Artif. Intell.* 206 (2014), pp. 25–52. DOI: [10.1016/j.artint.2013.09.006](https://doi.org/10.1016/j.artint.2013.09.006).
- [7] Andrew Canis, Stephen Dean Brown and Jason Helge Anderson. ‘Modulo SDC scheduling with recurrence minimization in high-level synthesis’. In: *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*. IEEE, 2014, pp. 1–8. DOI: [10.1109/FPL.2014.6927490](https://doi.org/10.1109/FPL.2014.6927490).
- [8] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz S. Czajkowski, Stephen Dean Brown and Jason Helge Anderson. ‘LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems’. In: *ACM Trans. Embedded Comput. Syst.* 13.2 (2013), 24:1–24:27. DOI: [10.1145/2514740](https://doi.org/10.1145/2514740).

- [9] Adrian M. Caulfield et al. 'A cloud-scale acceleration architecture'. In: *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. IEEE Computer Society, 2016, 7:1-7:13. DOI: [10.1109/MICRO.2016.7783710](https://doi.org/10.1109/MICRO.2016.7783710).
- [10] Shenghsun Cho, Michael Ferdman and Peter Milder. 'FPGASwarm: High Throughput Model Checking on FPGAs'. In: *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*. IEEE Computer Society, 2018, pp. 435-442. DOI: [10.1109/FPL.2018.00080](https://doi.org/10.1109/FPL.2018.00080).
- [11] Josep M. Codina, Josep Llosa and Antonio González. 'A comparative study of modulo scheduling techniques'. In: *Proceedings of the 16th international conference on Supercomputing, ICS 2002, New York City, NY, USA, June 22-26, 2002*. Ed. by Kemal Ebcioglu, Keshav Pingali and Alex Nicolau. ACM, 2002, pp. 97-106. DOI: [10.1145/514191.514208](https://doi.org/10.1145/514191.514208).
- [12] Jason Cong, Muhuan Huang, Bin Liu, Peng Zhang and Yi Zou. 'Combining module selection and replication for throughput-driven streaming programs'. In: *2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012, Dresden, Germany, March 12-16, 2012*. Ed. by Wolfgang Rosenstiel and Lothar Thiele. IEEE, 2012, pp. 1018-1023. DOI: [10.1109/DATE.2012.6176645](https://doi.org/10.1109/DATE.2012.6176645).
- [13] Jason Cong, Muhuan Huang and Peng Zhang. 'Combining computation and communication optimizations in system synthesis for streaming applications'. In: *The 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '14, Monterey, CA, USA - February 26 - 28, 2014*. Ed. by Vaughn Betz and George A. Constantinides. ACM, 2014, pp. 213-222. DOI: [10.1145/2554688.2554771](https://doi.org/10.1145/2554688.2554771).
- [14] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees A. Vissers and Zhiru Zhang. 'High-Level Synthesis for FPGAs: From Prototyping to Deployment'. In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 30.4 (2011), pp. 473-491. DOI: [10.1109/TCAD.2011.2110592](https://doi.org/10.1109/TCAD.2011.2110592).
- [15] Jason Cong and Zhiru Zhang. 'An efficient and versatile scheduling algorithm based on SDC formulation'. In: *Proceedings of the 43rd Design Automation Conference, DAC 2006, San Francisco, CA, USA, July 24-28, 2006*. Ed. by Ellen Sentovich. ACM, 2006, pp. 433-438. DOI: [10.1145/1146909.1147025](https://doi.org/10.1145/1146909.1147025).
- [16] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman and F. Kenneth Zadeck. 'Efficiently Computing Static Single Assignment Form and the Control Dependence Graph'. In: *ACM Trans. Program. Lang. Syst.* 13.4 (1991), pp. 451-490. DOI: [10.1145/115372.115320](https://doi.org/10.1145/115372.115320).

- [17] Steve Dai and Zhiru Zhang. 'Improving Scalability of Exact Modulo Scheduling with Specialized Conflict-Driven Learning'. In: *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*. ACM, 2019, p. 127. DOI: [10.1145/3316781.3317842](https://doi.org/10.1145/3316781.3317842).
- [18] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. Electrical and Computer Engineering Series. McGraw-Hill, 1994. ISBN: 9780070163331.
- [19] Benoît Dupont de Dinechin. *Simplex Scheduling: More than Lifetime-Sensitive Instruction Scheduling*. Tech. rep. PRISM 1994.22. 1994.
- [20] Benoît Dupont de Dinechin. 'Time-Indexed Formulations and a Large Neighborhood Search for the Resource-Constrained Modulo Scheduling Problem'. In: *Proceedings of the 3rd Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA 2007), 28-31 August 2007, Paris, France*. Ed. by P. Baptiste, G. Kendall, A. Munier-Kordon and F. Sourd. 2007, pp. 144–151.
- [21] Elizabeth D. Dolan and Jorge J. Moré. 'Benchmarking optimization software with performance profiles'. In: *Math. Program.* 91.2 (2002), pp. 201–213. DOI: [10.1007/s101070100263](https://doi.org/10.1007/s101070100263).
- [22] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: <http://archive.ics.uci.edu/ml>.
- [23] M. Ehrgott. *Multicriteria Optimization*. Springer Berlin Heidelberg, 2010. ISBN: 9783642059759. URL: <https://books.google.de/books?id=ruiKcgAACAAJ>.
- [24] Alexandre E. Eichenberger. 'Modulo Scheduling, Machine Representations, and Register-sensitive Algorithms'. AAI9711956. PhD thesis. Ann Arbor, MI, USA, 1996. ISBN: 0-591-19499-6.
- [25] Alexandre E. Eichenberger and Edward S. Davidson. 'A Reduced Multipipeline Machine Description that Preserves Scheduling Constraints'. In: *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadelphia, Pennsylvania, USA, May 21-24, 1996*. Ed. by Charles N. Fischer. ACM, 1996, pp. 12–22. DOI: [10.1145/231379.231386](https://doi.org/10.1145/231379.231386).
- [26] Alexandre E. Eichenberger and Edward S. Davidson. 'Efficient Formulation for Optimal Modulo Schedulers'. In: *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI), Las Vegas, Nevada, USA, June 15-18, 1997*. Ed. by Marina C. Chen, Ron K. Cytron and A. Michael Berman. ACM, 1997, pp. 194–205. DOI: [10.1145/258915.258933](https://doi.org/10.1145/258915.258933).

- [27] Alexandre E. Eichenberger, Edward S. Davidson and Santosh G. Abraham. 'Optimum Modulo Schedules for Minimum Register Requirements'. In: *Proceedings of the 9th international conference on Supercomputing, ICS 1995, Barcelona, Spain, July 3-7, 1995*. Ed. by Mateo Valero. ACM, 1995, pp. 31–40. DOI: [10.1145/224538.224542](https://doi.org/10.1145/224538.224542).
- [28] Alexandre E. Eichenberger, Edward S. Davidson and Santosh G. Abraham. 'Author retrospective for optimum modulo schedules for minimum register requirements'. In: *ACM International Conference on Supercomputing 25th Anniversary Volume*. Ed. by Utpal Banerjee. ACM, 2014, pp. 35–36. DOI: [10.1145/2591635.2591653](https://doi.org/10.1145/2591635.2591653).
- [29] Kevin Fan, Manjunath Kudlur, Hyunchul Park and Scott A. Mahlke. 'Cost Sensitive Modulo Scheduling in a Loop Accelerator Synthesis System'. In: *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-38 2005), 12-16 November 2005, Barcelona, Spain*. IEEE Computer Society, 2005, pp. 219–232. DOI: [10.1109/MICRO.2005.17](https://doi.org/10.1109/MICRO.2005.17).
- [30] Fabrizio Ferrandi, Pier Luca Lanzi, Daniele Loiacono, Christian Pilato and Donatella Sciuto. 'A Multi-objective Genetic Algorithm for Design Space Exploration in High-Level Synthesis'. In: *IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2008, 7-9 April 2008, Montpellier, France*. IEEE Computer Society, 2008, pp. 417–422. DOI: [10.1109/ISVLSI.2008.73](https://doi.org/10.1109/ISVLSI.2008.73).
- [31] Dirk Fimmel and Jan Müller. 'Optimal Software Pipelining with Rational Initiation Interval'. In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '02, June 24 - 27, 2002, Las Vegas, Nevada, USA, Volume 2*. Ed. by Hamid R. Arabnia. CSREA Press, 2002, pp. 638–643.
- [32] Robert W. Floyd. 'Algorithm 97: Shortest path'. In: *Commun. ACM* 5.6 (1962), p. 345. DOI: [10.1145/367766.368168](https://doi.org/10.1145/367766.368168).
- [33] GCC, the GNU compiler collection. 2019. URL: <https://gcc.gnu.org>.
- [34] Gina Goff, Ken Kennedy and Chau-Wen Tseng. 'Practical Dependence Testing'. In: *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*. Ed. by David S. Wise. ACM, 1991, pp. 15–29. DOI: [10.1145/113445.113448](https://doi.org/10.1145/113445.113448).
- [35] Rajesh Gupta and Forrest Brewer. 'High-Level Synthesis: A Retrospective'. In: *High-Level Synthesis: From Algorithm to Digital Circuit*. Ed. by Philippe Coussy and Adam Morawiec. Dordrecht: Springer Netherlands, 2008, pp. 13–28. ISBN: 978-1-4020-8588-8. DOI: [10.1007/978-1-4020-8588-8_2](https://doi.org/10.1007/978-1-4020-8588-8_2).

- [36] *Gurobi Documentation: Constraints*. 2019. URL: <https://www.gurobi.com/documentation/8.1/refman/constraints.html>.
- [37] Horst W. Hamacher and Kathrin Klamroth. *Lineare und Netzwerk-Optimierung - ein bilinguales Lehrbuch: a bilingual textbook*. Vieweg, 2000. ISBN: 978-3-528-03155-8.
- [38] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda and Hiroaki Takada. 'Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis'. In: *JIP* 17 (2009), pp. 242–254. DOI: [10.2197/ipsjjip.17.242](https://doi.org/10.2197/ipsjjip.17.242).
- [39] Jaco Hofmann, Jens Korinth and Andreas Koch. 'A Scalable High-Performance Hardware Architecture for Real-Time Stereo Vision by Semi-Global Matching'. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2016, Las Vegas, NV, USA, June 26 - July 1, 2016*. IEEE Computer Society, 2016, pp. 845–853. DOI: [10.1109/CVPRW.2016.110](https://doi.org/10.1109/CVPRW.2016.110).
- [40] Richard A. Huff. 'Lifetime-Sensitive Modulo Scheduling'. In: *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*. Ed. by Robert Cartwright. ACM, 1993, pp. 258–267. DOI: [10.1145/155090.155115](https://doi.org/10.1145/155090.155115).
- [41] Jens Huthmann. 'An Execution Model and High-Level-Synthesis System for Generating SIMT Multi-Threaded Hardware from C Source Code'. PhD thesis. Darmstadt University of Technology, Germany, 2017. URL: <http://tuprints.ulb.tu-darmstadt.de/6776/>.
- [42] Jens Huthmann, Björn Liebig, Julian Oppermann and Andreas Koch. 'Hardware/software co-compilation with the Nymble system'. In: *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), Darmstadt, Germany, July 10-12, 2013*. IEEE, 2013, pp. 1–8. DOI: [10.1109/ReCoSoC.2013.6581538](https://doi.org/10.1109/ReCoSoC.2013.6581538).
- [43] Cheng-Tsung Hwang, Jiahn-Hung Lee and Yu-Chin Hsu. 'A formal approach to the scheduling problem in high level synthesis'. In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 10.4 (1991), pp. 464–475. DOI: [10.1109/43.75629](https://doi.org/10.1109/43.75629).
- [44] Zheming Jin and Hal Finkel. 'Evaluating LULESH Kernels on OpenCL FPGA'. In: *Applied Reconfigurable Computing - 15th International Symposium, ARC 2019, Darmstadt, Germany, April 9-11, 2019, Proceedings*. Ed. by Christian Hochberger, Brent Nelson, Andreas Koch, Roger F. Woods and Pedro C. Diniz. Vol. 11444. Lecture Notes in Computer Science. Springer, 2019, pp. 199–213. DOI: [10.1007/978-3-030-17227-5_15](https://doi.org/10.1007/978-3-030-17227-5_15).

- [45] Ian Karlin, Jeff Keasler and Rob Neely. *LULESH 2.0 Updates and Changes*. Tech. rep. LLNL-TR-641973. Livermore, CA, Aug. 2013, pp. 1–9.
- [46] Ryan Kastner, Janarбек Matai and Stephen Neuendorffer. ‘Parallel Programming for FPGAs’. In: *CoRR* abs/1805.03648 (2018). arXiv: 1805.03648. URL: <http://arxiv.org/abs/1805.03648>.
- [47] Khronos Group: *OpenCL Overview*. 2019. URL: <https://www.khronos.org/opencl/>.
- [48] Ed Klotz and Alexandra M. Newman. ‘Practical guidelines for solving difficult mixed integer linear programs’. In: *Surveys in Operations Research and Management Science* 18.1 (2013), pp. 18–32. ISSN: 1876–7354. DOI: [10.1016/j.sorms.2012.12.001](https://doi.org/10.1016/j.sorms.2012.12.001).
- [49] Jens Korinth, David de la Chevallierie and Andreas Koch. ‘An Open-Source Tool Flow for the Composition of Reconfigurable Hardware Thread Pool Architectures’. In: *23rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2015, Vancouver, BC, Canada, May 2-6, 2015*. IEEE Computer Society, 2015, pp. 195–198. DOI: [10.1109/FCCM.2015.22](https://doi.org/10.1109/FCCM.2015.22).
- [50] Jens Korinth, Jaco Hofmann, Carsten Heinz and Andreas Koch. ‘The TaPaSCo Open-Source Toolflow for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems’. In: *Applied Reconfigurable Computing - 15th International Symposium, ARC 2019, Darmstadt, Germany, April 9-11, 2019, Proceedings*. Ed. by Christian Hochberger, Brent Nelson, Andreas Koch, Roger F. Woods and Pedro C. Diniz. Vol. 11444. Lecture Notes in Computer Science. Springer, 2019, pp. 214–229. DOI: [10.1007/978-3-030-17227-5_16](https://doi.org/10.1007/978-3-030-17227-5_16).
- [51] Manjunath Kudlur, Kevin Fan and Scott A. Mahlke. ‘Streamroller: : automatic synthesis of prescribed throughput accelerator pipelines’. In: *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2006, Seoul, Korea, October 22-25, 2006*. Ed. by Reinaldo A. Bergamaschi and Kiyong Choi. ACM, 2006, pp. 270–275. DOI: [10.1145/1176254.1176321](https://doi.org/10.1145/1176254.1176321).
- [52] Monica S. Lam. ‘Software Pipelining: An Effective Scheduling Technique for VLIW Machines’. In: *Proceedings of the ACM SIGPLAN’88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*. Ed. by Richard L. Wexelblat. ACM, 1988, pp. 318–328. DOI: [10.1145/53990.54022](https://doi.org/10.1145/53990.54022).
- [53] Chris Lattner and Vikram S. Adve. ‘LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation’. In: *2nd IEEE / ACM International Symposium on Code Generation*

- and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA. IEEE Computer Society, 2004, pp. 75–88. DOI: [10.1109/CGO.2004.1281665](https://doi.org/10.1109/CGO.2004.1281665).
- [54] Peng Li, Peng Zhang, Louis-Noël Pouchet and Jason Cong. ‘Resource-Aware Throughput Optimization for High-Level Synthesis’. In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 22-24, 2015*. Ed. by George A. Constantinides and Deming Chen. ACM, 2015, pp. 200–209. DOI: [10.1145/2684746.2689065](https://doi.org/10.1145/2684746.2689065).
 - [55] Björn Liebig. ‘Domain-Specific High Level Synthesis of Floating-Point Computations to Resource-Shared Microarchitectures’. PhD thesis. Darmstadt University of Technology, Germany, 2018. URL: <http://tuprints.ulb.tu-darmstadt.de/7338/>.
 - [56] Josep Llosa, Eduard Ayguadé, Antonio González, Mateo Valero and Jason Eckhardt. ‘Lifetime-Sensitive Modulo Scheduling in a Production Environment’. In: *IEEE Trans. Computers* 50.3 (2001), pp. 234–249. DOI: [10.1109/12.910814](https://doi.org/10.1109/12.910814).
 - [57] Andrea Lodi and Andrea Tramontani. ‘Performance Variability in Mixed-Integer Programming’. In: *Theory Driven by Influential Applications*. Chap. Chapter 1, pp. 1–12. DOI: [10.1287/educ.2013.0112](https://doi.org/10.1287/educ.2013.0112).
 - [58] Liqiang Lu, Jiaming Xie, Ruirui Huang, Jiansong Zhang, Wei Lin and Yun Liang. ‘An Efficient Hardware Accelerator for Sparse Convolutional Neural Networks on FPGAs’. In: *27th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2019, San Diego, CA, USA, April 28 - May 1, 2019*. IEEE, 2019, pp. 17–25. DOI: [10.1109/FCCM.2019.00013](https://doi.org/10.1109/FCCM.2019.00013).
 - [59] Scott A. Mahlke, Richard E. Hank, Roger A. Bringmann, John C. Gyllenhaal, David M. Gallagher and Wen-mei W. Hwu. ‘Characterizing the impact of predicated execution on branch prediction’. In: *Proceedings of the 27th Annual International Symposium on Microarchitecture, San Jose, California, USA, November 30 - December 2, 1994*. Ed. by Hans Mulder and Matthew K. Farrens. ACM / IEEE Computer Society, 1994, pp. 217–227. DOI: [10.1109/MICRO.1994.717461](https://doi.org/10.1109/MICRO.1994.717461).
 - [60] Alejandro Molina, Antonio Vergari, Nicola Di Mauro, Sriraam Natarajan, Floriana Esposito and Kristian Kersting. ‘Mixed Sum-Product Networks: A Deep Architecture for Hybrid Domains’. In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. Ed. by Sheila A. McIlraith and Kilian Q. Weinberger. AAAI Press, 2018, pp. 3828–3835. URL:

<https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16865>.

- [61] Razvan Nane et al. 'A Survey and Evaluation of FPGA High-Level Synthesis Tools'. In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 35.10 (2016), pp. 1591–1604. DOI: [10.1109/TCAD.2015.2513673](https://doi.org/10.1109/TCAD.2015.2513673).
- [62] Brent E. Nelson. 'Third Party CAD Tools for FPGA Design - A Survey of the Current Landscape'. In: *Applied Reconfigurable Computing - 15th International Symposium, ARC 2019, Darmstadt, Germany, April 9-11, 2019, Proceedings*. Ed. by Christian Hochberger, Brent Nelson, Andreas Koch, Roger F. Woods and Pedro C. Diniz. Vol. 11444. Lecture Notes in Computer Science. Springer, 2019, pp. 353–367. DOI: [10.1007/978-3-030-17227-5_25](https://doi.org/10.1007/978-3-030-17227-5_25).
- [63] Julian Oppermann, Andreas Koch, Melanie Reuter-Oppermann and Oliver Sinnen. 'ILP-based modulo scheduling for high-level synthesis'. In: *2016 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2016, Pittsburgh, Pennsylvania, USA, October 1-7, 2016*. ACM, 2016, 1:1–1:10. DOI: [10.1145/2968455.2968512](https://doi.org/10.1145/2968455.2968512).
- [64] Julian Oppermann, Andreas Koch, Ting Yu and Oliver Sinnen. 'Domain-specific optimisation for the high-level synthesis of CellML-based simulation accelerators'. In: *25th International Conference on Field Programmable Logic and Applications, FPL 2015, London, United Kingdom, September 2-4, 2015*. IEEE, 2015, pp. 1–7. DOI: [10.1109/FPL.2015.7294019](https://doi.org/10.1109/FPL.2015.7294019).
- [65] Julian Oppermann, Melanie Reuter-Oppermann, Lukas Sommer, Andreas Koch and Oliver Sinnen. 'Exact and Practical Modulo Scheduling for High-Level Synthesis'. In: *TRETS 12.2* (2019), 8:1–8:26. DOI: [10.1145/3317670](https://doi.org/10.1145/3317670).
- [66] Julian Oppermann, Melanie Reuter-Oppermann, Lukas Sommer, Oliver Sinnen and Andreas Koch. 'Dependence Graph Preprocessing for Faster Exact Modulo Scheduling in High-Level Synthesis'. In: *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*. IEEE Computer Society, 2018, pp. 280–286. DOI: [10.1109/FPL.2018.00055](https://doi.org/10.1109/FPL.2018.00055).
- [67] Julian Oppermann, Patrick Sittel, Martin Kumm, Melanie Reuter-Oppermann, Andreas Koch and Oliver Sinnen. 'Design-Space Exploration with Multi-Objective Resource-Aware Modulo Scheduling'. In: *Euro-Par 2019: Parallel Processing - 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26-30, 2019, Proceedings*. Ed. by Ramin Yahyapour. Vol. 11725. Lecture Notes in Computer Science. Springer, 2019, pp. 170–183. DOI: [10.1007/978-3-030-29400-7_13](https://doi.org/10.1007/978-3-030-29400-7_13).

- [68] Julian Oppermann, Lukas Sommer, Lukas Weber, Melanie Reuter-Oppermann, Andreas Koch and Oliver Sinnen. ‘SkyCastle: A Resource-Aware Multi-Loop Scheduler for High-Level Synthesis’. In: *International Conference on Field-Programmable Technology, FPT 2019*. 2019.
- [69] Julian Oppermann, Sebastian Vollbrecht, Melanie Reuter-Oppermann, Oliver Sinnen and Andreas Koch. ‘GeMS: a generator for modulo scheduling problems: work in progress’. In: *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES 2018, Torino, Italy, September 30 - October 05, 2018*. Ed. by Tulika Mitra and Akash Kumar. ACM, 2018, 7:1–7:3. URL: <http://dl.acm.org/citation.cfm?id=3283559>.
- [70] *Origami HLS*. 2019. URL: <http://www.uni-kassel.de/go/origami>.
- [71] Christian Pilato and Fabrizio Ferrandi. ‘Bambu: A modular framework for the high level synthesis of memory-intensive applications’. In: *23rd International Conference on Field programmable Logic and Applications, FPL 2013, Porto, Portugal, September 2-4, 2013*. IEEE, 2013, pp. 1–4. DOI: [10.1109/FPL.2013.6645550](https://doi.org/10.1109/FPL.2013.6645550).
- [72] Hoifung Poon and Pedro M. Domingos. ‘Sum-Product Networks: A New Deep Architecture’. In: *UAI 2011, Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence, Barcelona, Spain, July 14-17, 2011*. Ed. by Fábio Gagliardi Cozman and Avi Pfeffer. AUAI Press, 2011, pp. 337–346. URL: https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=2194&proceeding_id=27.
- [73] Adrien Prost-Boucle, Olivier Muller and Frédéric Rousseau. ‘Fast and standalone Design Space Exploration for High-Level Synthesis under resource constraints’. In: *Journal of Systems Architecture - Embedded Systems Design* 60.1 (2014), pp. 79–93. DOI: [10.1016/j.sysarc.2013.10.002](https://doi.org/10.1016/j.sysarc.2013.10.002).
- [74] B. Ramakrishna Rau. ‘Iterative Modulo Scheduling’. In: *International Journal of Parallel Programming* 24.1 (1996), pp. 3–65.
- [75] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei and David M. Brooks. ‘MachSuite: Benchmarks for accelerator design and customized architectures’. In: *2014 IEEE International Symposium on Workload Characterization, IISWC 2014, Raleigh, NC, USA, October 26-28, 2014*. IEEE Computer Society, 2014, pp. 110–119. DOI: [10.1109/IISWC.2014.6983050](https://doi.org/10.1109/IISWC.2014.6983050).
- [76] John C. Rutenber, Guang R. Gao, Woody Lichtenstein and Artour Stoutchinin. ‘Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler’. In: *Proceedings of the ACM SIGPLAN’96 Conference on Programming Language Design and Implementation (PLDI), Philadelphia, Pennsylvania, USA,*

- May 21-24, 1996. Ed. by Charles N. Fischer. ACM, 1996, pp. 1–11. DOI: [10.1145/231379.231385](https://doi.org/10.1145/231379.231385).
- [77] Benjamin Carrión Schäfer. ‘Probabilistic Multiknob High-Level Synthesis Design Space Exploration Acceleration’. In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 35.3 (2016), pp. 394–406. DOI: [10.1109/TCAD.2015.2472007](https://doi.org/10.1109/TCAD.2015.2472007).
 - [78] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley Series in Discrete Mathematics & Optimization. Wiley, 1998. ISBN: 9780471982326.
 - [79] Patrick Sittel, Martin Kumm, Julian Oppermann, Konrad Möller, Peter Zipf and Andreas Koch. ‘ILP-Based Modulo Scheduling and Binding for Register Minimization’. In: *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*. IEEE Computer Society, 2018, pp. 265–271. DOI: [10.1109/FPL.2018.00053](https://doi.org/10.1109/FPL.2018.00053).
 - [80] Patrick Sittel, Julian Oppermann, Martin Kumm, Andreas Koch and Peter Zipf. ‘HatScheT: A Contribution to Agile HLS’. In: *International Workshop on FPGAs for Software Programmers (FSP)*. 2018, pp. 1–8.
 - [81] Patrick Sittel, John Wickerson, Martin Kumm and Peter Zipf. ‘Modulo Scheduling with Rational Initiation Intervals in Custom Hardware Design’. In: *Proceedings of the 25th Asia and South Pacific Design Automation Conference, ASPDAC 2020, Beijing, China*. 2020.
 - [82] Leonardo Solis-Vasquez and Andreas Koch. ‘A Case Study in Using OpenCL on FPGAs: Creating an Open-Source Accelerator of the AutoDock Molecular Docking Software’. In: *Fifth International Workshop on FPGAs for Software Programmers (FSP)*. 2018.
 - [83] Lukas Sommer, Julian Oppermann, Alejandro Molina, Carsten Binnig, Kristian Kersting and Andreas Koch. ‘Automatic Mapping of the Sum-Product Network Inference Problem to FPGA-Based Accelerators’. In: *36th IEEE International Conference on Computer Design, ICCD 2018, Orlando, FL, USA, October 7-10, 2018*. IEEE Computer Society, 2018, pp. 350–357. DOI: [10.1109/ICCD.2018.00060](https://doi.org/10.1109/ICCD.2018.00060).
 - [84] Leandro de Souza Rosa, Christos-Savvas Bouganis and Vanderlei Bonato. ‘Scaling Up Modulo Scheduling for High-Level Synthesis’. In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 38.5 (2019), pp. 912–925. DOI: [10.1109/TCAD.2018.2834440](https://doi.org/10.1109/TCAD.2018.2834440).
 - [85] James Stanier and Des Watson. ‘Intermediate representations in imperative compilers: A survey’. In: *ACM Comput. Surv.* 45.3 (2013), 26:1–26:27. DOI: [10.1145/2480741.2480743](https://doi.org/10.1145/2480741.2480743).

- [86] Přemysl Šůcha and Zdeněk Hanzálek. ‘A cyclic scheduling problem with an undetermined number of parallel identical processors’. In: *Comp. Opt. and Appl.* 48.1 (2011), pp. 71–90. DOI: [10.1007/s10589-009-9239-4](https://doi.org/10.1007/s10589-009-9239-4).
- [87] Welson Sun, Michael J. Wirthlin and Stephen Neuendorffer. ‘FPGA Pipeline Synthesis Design Exploration Using Module Selection and Resource Sharing’. In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 26.2 (2007), pp. 254–265. DOI: [10.1109/TCAD.2006.887923](https://doi.org/10.1109/TCAD.2006.887923).
- [88] Ole Tange. ‘GNU Parallel - The Command-Line Power Tool’. In: *login: The USENIX Magazine* 36.1 (Feb. 2011), pp. 42–47. URL: <http://www.gnu.org/s/parallel>.
- [89] Bram Veenboer and John W. Romein. ‘Radio-Astronomical Imaging: FPGAs vs GPUs’. In: *Euro-Par 2019: Parallel Processing - 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26-30, 2019, Proceedings*. Ed. by Ramin Yahyapour. Vol. 11725. Lecture Notes in Computer Science. Springer, 2019, pp. 509–521. DOI: [10.1007/978-3-030-29400-7_36](https://doi.org/10.1007/978-3-030-29400-7_36).
- [90] Sarad Venugopalan and Oliver Sinnen. ‘ILP Formulations for Optimal Task Scheduling with Communication Delays on Parallel Systems’. In: *IEEE Trans. Parallel Distrib. Syst.* 26.1 (2015), pp. 142–151. DOI: [10.1109/TPDS.2014.2308175](https://doi.org/10.1109/TPDS.2014.2308175).
- [91] *Vivado High-Level Synthesis*. 2019. URL: <https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html>.
- [92] Haomiao Wang, P. Thiagaraj and O. Sinnen. ‘Combining Multiple Optimised FPGA-based Pulsar Search Modules Using OpenCL’. In: *Journal of Astronomical Instrumentation* (2019).
- [93] H.S. Warren. *Hacker’s Delight*. Addison-Wesley, 2013. ISBN: 9780321842688.
- [94] L.A. Wolsey. *Integer Programming*. Wiley Series in Discrete Mathematics and Optimization. Wiley, 1998. ISBN: 9780471283669.
- [95] Zhiru Zhang and Bin Liu. ‘SDC-based modulo scheduling for pipeline synthesis’. In: *The IEEE/ACM International Conference on Computer-Aided Design, ICCAD’13, San Jose, CA, USA, November 18-21, 2013*. Ed. by Jörg Henkel. IEEE, 2013, pp. 211–218. DOI: [10.1109/ICCAD.2013.6691121](https://doi.org/10.1109/ICCAD.2013.6691121).
- [96] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang and Bingsheng He. ‘COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications’. In: *2017 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2017, Irvine, CA, USA, November 13-16, 2017*. Ed. by Sri Parameswaran. IEEE, 2017, pp. 430–437. DOI: [10.1109/ICCAD.2017.8203809](https://doi.org/10.1109/ICCAD.2017.8203809).

- [97] Ritchie Zhao, Mingxing Tan, Steve Dai and Zhiru Zhang. 'Area-efficient pipelining for FPGA-targeted high-level synthesis'. In: *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*. ACM, 2015, 157:1–157:6. DOI: [10.1145/2744769.2744801](https://doi.org/10.1145/2744769.2744801).
- [98] Guanwen Zhong, Alok Prakash, Yun Liang, Tulika Mitra and Smaïl Niar. 'Lin-analyzer: a high-level performance analysis tool for FPGA-based accelerators'. In: *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*. ACM, 2016, 136:1–136:6. DOI: [10.1145/2897937.2898040](https://doi.org/10.1145/2897937.2898040).